Universidad Carlos III de Madrid
Escuela Politécnica Superior
Departamento de Informática

Tesis Doctoral

Técnicas de optimización dinámicas de aplicaciones paralelas basadas en MPI

Ph.D. Thesis

Dynamic optimization techniques to enhance scalability and performance of MPI-based applications

Author: Rosa Filgueira Vicente
Advisors: Jesús Carretero Pérez
David Expósito Singh

Leganés, 2010

TESIS DOCTORAL

Técnicas de optimización dinámicas de aplicaciones paralelas basadas en MPI

AUTOR: **Rosa Filgueira Vicente**
DIRECTORES: **Jesús Carretero Pérez**
David Expósito Singh

TRIBUNAL CALIFICADOR

PRESIDENTE:

VOCALES:

VOCAL SECRETARIO:

CALIFICACION:

Leganés, a de de 2010

A mis padres, a mi familia y
amigos.

Agradecimientos

Después de tantos años de trabajo me parece mentira, que por fin este escribiendo los agradecimientos de tesis. Me gustaría empezar resaltando que una tesis no solo es fruto del esfuerzo del autor y de los directores de la tesis, si no también, de todas las personas que te acompañan durante la realización de la misma, ya que voluntariamente o involuntariamente se ven implicadas en ella: Padres, familia, amigos, compañeros ... Por ello, los siguientes párrafos, van dedicados a vosotros.

A mis directores de tesis, Jesús y David. Para mi es todo un honor haber realizado este trabajo bajo la dirección de ambos. Os estaré siempre muy agradecida por vuestro tiempo y esfuerzo para sacar este trabajo adelante. Quiero aprovechar, para dar especialmente las gracias a Jesús, por darme la oportunidad de entrar en ARCOS y así poder trabajar en lo que más me gusta, investigar.

A mis padres, por vuestra paciencia, dedicación, apoyo y cariño incondicional. Sin vosotros sería imposible haber llegado hasta aquí. Me habéis ayudado a seguir adelante. Habéis estado conmigo siempre, tanto en los buenos como en los malos momentos. Cuantas llamadas de teléfono, cuantos viajes a Madrid, cuantas horas hablando ... Esta tesis es tan mía como vuestra. Muchísmas gracias, mama y papa.

A Cana, por ser mi gran compañero de despacho y amigo. Por todos los grandísimos momentos que hemos vivido tanto dentro como fuera de la Universidad. No son pocas las horas de trabajo, risas, viajes, dolores de cabeza, fiestas ... que hemos pasado juntos. Tengo anécdotas para escribir un libro entero. Has hecho que todos estos años sean muchísimo más divertidos, y eso es esencial para mí. Simplemente, gracias por todo. Como voy a echar de menos ver ese flequillo detras del monitor el año que viene.

A Pichel, otro de mis grandes pilares. No tengo palabras para agradecerte todo lo que has hecho por mi. Creíste en mi (y en mi tesis) sin apenas conocerme, y aún sigues haciéndolo. Contigo también son innumerables los buenos momentos que hemos pasado tanto dentro como fuera del trabajo. Gracias por estar siempre cuando te necesito. Eres todo un ejemplo a seguir en multitud de aspectos para mi. Estos dos últimos años, se te ha echado muchísimo de menos. Y no solo yo, las comidas de los viernes ya no son tan culturetas como antes. Eso si, en tu honor, seguimos tomando el cafe sentados. Moitas grazas.

A Isra y Jose, que os conocí el primer día de "clase" de Doctorado. Os empecé a hablar desde el minuto uno, y desde entonces no he callado. Aún recuerdo todas las preguntas de doctorado que os hacía sin conoceros de nada, y hoy puedo presumir de tener dos buenos amigos. Gracias por todas las veces que me habeís ayudado (eso incluye el poster, Isra). Os deseo lo mejor.

A Luismi, Javi "Doc", Alex y Roberto, que me habeís ayudado desde que entre en ARCOS y siempre con una sonrisa. He aprendido mucho de todos y cada uno de vosotros, y encima habeís logrado convertirme en una pseudo-friki!! Tener claro, que sin vosotros nunca hubiese terminado esta tesis. Ahora el jueves, siempre será el día del cine. Ha sido un gran placer, haberos conocido.

A Marian, que gran día cuando nos reencontramos. Desde entonces, has sido una buenísima amiga y un gran apoyo para mi. Me ha encantado pasar este año

contigo compartiendo agujetas y muchas más cosas, claro. Gracias por ser tal y como eres. Me da mucha pena ahora que nos vamos a separar, pero como se que eres una estupenda persona, amiga, e Ingeniera, no dudo que todo te va ir genial.

A mis amigas de Noia, Miriam, Angela y Bego, por estar siempre ahí desde hace tanto tiempo. Sois mis amigas de toda la vida, y aunque solo nos vemos en vacaciones, me acuerdo de vosotras durante todo el año. Gracias por todos los momentos que hemos pasado (y aún pasaremos muchos más) juntas en el Trabu, en los fines de año, en multitud de churrascadas, en la Aguieira, en los Apostoles, en el Jara, en mi querida Feria Medieval, ... la lista es interminable. Por vosotras me encanta Noia.

A todo el resto de amigos que he ido conociendo durante todos estos años en Madrid como Isa, Ana, Laura, Paco, Luis, amigos de residencia, de Master, etc. Sois muchísmos. Gracias por todo el apoyo, ánimo y sobre todo cariño y amistad.

Así también, a todos mis compañeros que forman o han formado parte de ARCOS.

A todos, muchisímas gracias.

Summary

Parallel computation on cluster architectures has become the most common solution for developing high-performance scientific applications. Message Passing Interface (MPI) [Mes94] is the message-passing library most widely used to provide communications in clusters. MPI provides a standard interface for operations such as point-to-point communication, collective communication, synchronization, and I/O operations.

Along the I/O phase, the processes frequently access a common data set by issuing a large number of small non-contiguous I/O requests [NKP+96a, SR98], which might create bottlenecks in the I/O subsystem. These bottlenecks are still higher in commodity clusters, where commercial networks are usually installed. Many of those networks, such as Fast Ethernet or Gigabit, have high latency and low bandwidth which introduce performance penalties during the program execution.

Scalability is also an important issue in cluster systems when many processors are used, which may cause network saturation and still higher latencies. As communication-intensive parallel applications spend a significant amount of their total execution time exchanging data between processes, the former problems may lead to poor performance not only in the I/O subsystem, but also in communication phase. Therefore, we can conclude that it is necessary to develop techniques for improving the performance of both communication and I/O subsystems.

The main goal of this Ph.D. thesis is to improve the scalability and performance of MPI-based applications executed in clusters reducing the overhead of I/O and communications subsystems. In summary, this work proposes two techniques that solve these problems in an efficient way managing the high complexity of a heterogeneous environment:

- **Reduction in the number of communications in collective I/O operations:** This thesis targets the reduction of the bottleneck in the I/O subsystem. Many applications use collective I/O operations to read/write data from/to disk. One of the most used is the *Two-Phase I/O* technique extended by Thakur and Choudhary in *ROMIO*. In this technique, many communications among the processes are performed, which could create a bottleneck. This bottleneck is still higher in commodity clusters, where commercial networks are usually installed, and in CMP clusters where the I/O bus is shared by the cores of a single node. Therefore, we propose improving locality in order to reduce the number of communications performed in *Two-Phase I/O*.

- **Reduction of transferred data volume:** This thesis attemps to reduce the cost of interchanged messages, reducing the data volume by using lossless compression among processes. Furthermore, we propose turning compression on and off and selecting at run-time the most appropriate compression algorithms depending on the characteristics of each message, network performance, and compression algorithms behavior.

Resumen

En la actualidad, las aplicaciones utilizadas en los entornos de computación de altas prestaciones, como por ejemplo simulaciones científicas o aplicaciones dedicadas a la extracción de datos (*data-mining*), necesitan además de enormes recursos de cómputo y memoria, el manejo de ingentes volúmenes de información.

Las arquitecturas *cluster* se han convertido en la solución más común para ejecutar este tipo de aplicaciones. La librería MPI (Message Passing Interface) [Mes94] es la más utilizada en estos entornos, ya que ofrece un interfaz estándar para operaciones de comunicación punto a punto, colectivas, sincronización y de E/S.

Durante la fase de E/S de las aplicaciones, los procesos acceden a un gran conjunto de datos mediante pequeñas peticiones de datos no-contiguos, por lo que pueden provocar cuellos de botella en el sistema de E/S. Estos cuellos de botella, pueden ser todavía mayor en los cluster, ya que se suelen utilizar redes comerciales como Fast Ethernet o Gigabit, las cuales tienen una gran latencia y bajo ancho de banda.

Por otra parte la escalabilidad es un importante problema en los clusters, cuando se ejecutan a la vez un gran número de procesos, ya que pueden causar saturación de la red, y aumenar la latencia. Como consecuencia de una comunicación intensiva, las aplicaciones gastan mucho tiempo intercambiando información entre los procesos, provocando problemas tanto en el sistema de comunicación, como en el de E/S. Por lo tanto, podemos concluir que en un cluster los subsistemas de E/S y de comunicaciones representan uno de los principales elementos en los que conviene mejorar su rendimiento.

El principal objetivo de esta Tesis Doctoral es mejorar la escalabilidad y rendimientos de las aplicaciones MPI ejecutadas en arquitecturas cluster, reduciendo la sobrecarga de los sistemas de comunicación y de E/S. Como resumen, este trabajo propone dos técnicas para resolver estos problemas de forma eficiente:

- **Reducción del número de comunicaciones en la operaciones colectivas de E/S:** Esta tesis tiene como uno de sus objetivos reducir los cuellos de botella producidos en el sistema de E/S. Muchas aplicaciones científicas utilizan operaciones colectivas de E/S para leer/escribir datos desde/al disco. Una de las técnicas más utilizas es *Two-Phase I/O* ampliada por Thakur and Choudhary en *ROMIO*. En esta técnica se realizan muchas comunicaciones entre los procesos, por lo que pueden crear un cuello de botella. Este cuello de botella es aún mayor en los cluster que tiene instaladas redes comerciales, y en los clusters multicore donde el bus de E/S es compartido por todos los cores de un mismo nodo. Por lo tanto, nosotros proponemos aumentar la localidad y disminuir a la vez en número de comunicaciones que se producen en *Two-Phase I/O* para reducir los problemas de E/S en las arquitecturas cluster.

- **Reducción del volumen de datos en las comunicaciones:** Esta tesis propone reducir el coste de las comunicaciones utilizando técnicas de compresión sin perdida. Concretamente, proponemos activar y desactivar la compresión y elegir el algoritmo de compresión en tiempo de ejecución, dependiendo de las características de cada mensaje, de la red y del comportamiento de los algoritmos de compresión.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents an introduction of the Ph.D. thesis. The first section presents the context of the thesis, the second section relates the major trends in high performance computing, and the third details the main objectives of this work. Finally, the last section summarizes the structure of the rest of the document.

1.1 Introduction

Supercomputers are large systems that contain hundreds, sometimes thousands, of processors and gigabytes of memory which, in parallel, simultaneously run portions of large computer simulations, data analysis, and numerical modelling. They are used for scientific modelling, military simulations and planning, the rendering of film animation and special effects, and countless other data-crunching medical and research needs.

Although corporations such as Cray, IBM, and SGI continue to manufacturer supercomputer distributed systems, cluster computing, which is the alternative technology of grouping individual systems to accomplish the same goals, is making inroads in the supercomputer market and reducing the demand for these systems.

Cluster computing is the result of connecting many local computers (nodes) together via a high speed connection to provide a single shared resource. Its distributed processing system allows complex computations to run in parallel as the tasks are shared by the individual processors and memory.

The concept of cluster computing has been played with for decades and actually has roots in the large mainframe and supercomputers of the 60's and 70's.

As defined in [vdS09], the adoption of clusters, collections of workstations/PCs connected by a local network, has virtually exploded since the introduction of the first *Beowulf* cluster in 1994 [Phi00]. The attraction lies in the (potentially) low

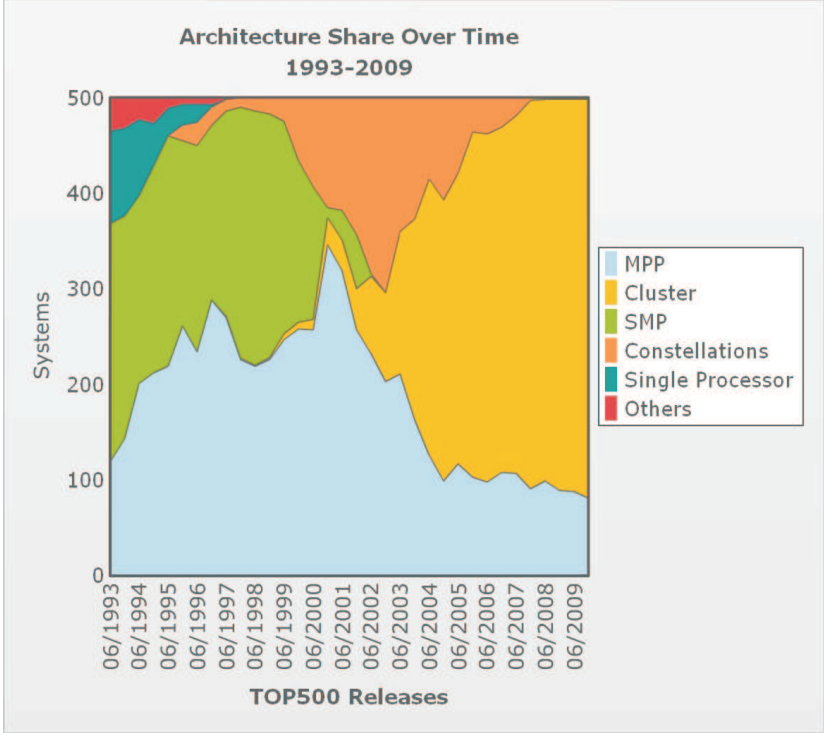


Figure 1.1: Evolution of architecture in HPC.

cost of both the hardware and software and the control that builders/users have over their system.

As we can see in Figure 1.1 clusters have become the most popular platform for High-Performance Computing (HPC). In addition, the emergence of multi-core architecture has brought clusters into a multi-core era. Multi-core means integrating two or more complete computational cores within a single chip. A multi-core processor is also referred to as Chip Multiprocessor (CMP).

Despite the maturity of cluster architecture, there are some critical challenges that need to be solved to improve the scalability and performance of applications. In [vdS09], two of them are cited and the third challenge is described in [DL08]:

- Network speed is very important in all but the most compute bound applications. Network technology has improved performance, achieving higher bandwidth and lower latency. But the cost of this high-end technology in large clusters converts this solution into a cost problem. Fortunately, Gigabit Ethernet or 10-Gigabit Ethernet is a very attractive solution for economic reasons. However, these kinds of networks very often suffer from congestion, degrading the global system performance. Therefore, it is necessary to develop software solutions to reduce network congestion as much as possible when commodity networks are installed in clusters.

- Another notable observation is that using compute nodes with more than 1 CPU (CMP) may be attractive from the point of view of compactness and

(possibly) energy and cooling aspects. Nevertheless, there are several factors that limit the performance of CMP clusters: One of the main important bottlenecks is I/O operations. In this way, I/O requests initiated by multiple cores may saturate the I/O bus. This fact becomes even more remarkable when issuing multiple non-contiguos disk accesses.

- Large-scale computing clusters with hundreds of tens of thousands of processors are being increasingly used to execute large, data-intensive applications in several scientific domains. The I/O requirements of such HPC applications can be staggering, ranging from terabytes to petabytes and beyond, and managing such massive data sets has created a significant bottleneck in application performance. Thus solving this I/O scalability problem has created a critical challenge in high-performance computing.

Hence, the work of this Ph.D. thesis is focused on proposing solutions for these very important challenges, in order to enhance the scalability and performance of applications executed in cluster environments.

To solve the the challenges presented, it is essential to determine the network, CPU and I/O information that the applications executed in those clusters give us. Applications usually access these elements by using MPI library [Mes94] which is the programming model most used in clusters. MPI provides a standard interface for operations such as point-to-point communication, collective communication, synchronization, and I/O operations. Therefore, in this Ph.D. thesis, we have choosen MPI as the best place to locate our solutions for these challenges.

1.2 Major trends in high performance computing

Over the last twenty years, the open source community has increasingly provided the software stacks at the heart of the world's HPC systems [DBA+09]. The community has invested a great deal of money and years of effort to build key components from low-level performance counter interfaces such as PAPI for the Linux operating system, GNU tools, MPI, math libraries such as PLASMA and PETSc, and new languages such as CoArray FORTRAN, UPC, and Fortress. However, while this investment is tremendously valuable, and at the core of all petascale machines, it is poorly coordinated, planned, and often missing key integration technologies. While open source development within a single project, such as MPICH, can be coordinated by a repository gatekeeper and a mailing list discussion, the community has no mechanism for identifying key holes in the software environment, integration areas, or coordination. With the explosion of multicore parallelism and new hardware models and features, such as transactional memory, speculative execution, and GPUs, this completely uncoordinated development model will not provide the needed software to support peta/exascale computation on millions of cores. Therefore, the International Exascale Software Project has been created to prepare for the challenges of exascale computing.

The mission of Exascale Software Project (IESP) is to create a plan for a coordinated international effort to create a common software cyber infrastructure capable of meeting the architectural challenges of current and future peta/exascale systems and the application challenges of current and future peta/exascale systems. For this purpose, the IESP held three workshops in the US, Europe, and Japan during 2009.

As follows, we summarize the topics and issues traded in the third IESP workshop [Moh09] related to our work:

- Intra-Node
  - **Locality and distributed data**: Locality of reference and some kind of global view of data objects are already important for large-scale parallel applications, and will become even more important. Remote memory access will become even more expensive and complex cache hierarchies will be difficult to manage. Presently, programming models have a variety of ways of expressing scope and locality, but there is no standard, nature environment that is suitable for exascale. Programming and application code levels, especially for global view programming. We need a telescoping approach to optimize data placement and motion, from automatic to selectively user controlled.
- Inter-Node
  - **I/O for Excascale**: Scalable I/O is identified as already critical for petascale exascale, with two major issues. One is programming and abstraction; The other is software performance and optimizations, in terms of achievable bandwidth and latency, including new software caching techniques as well as possibly exploiting new storage technologies such as SSD.
  - **Eliminate bottlenecks to strong scaling**: There are impeding problems that are already critical in today's petascale and will be most problematic for exascale. Many algorithms, either at the application level or systems level, embody O(N) (or beyond) portions, severe bottleneck when N = 1B (10^9). Scaling achieved in today's systems is weak scaling, but problem size enlargement, coupled with higher-order algorithmic requirements, will force strong scaling and a decrease in memory per core. This will be problematic, as most systems provide communication with high latency, resulting in systems that will to be able to exploit the 1B-way parallelism.

We wish to emphasize that the topics treated in this thesis are today in vogue in the scientific community, and they are currently being researched.

1.3 Main objectives of the Thesis

The main objective of this Ph.D. thesis is to improve the scalability and performance of MPI-based applications executed in clusters. In this kind of environment, the efficiency of parallel applications is maximized when the workload is evenly distributed among nodes and the overhead introduced in the parallelization process is minimized: the cost of communication, synchronization and I/O operations must be kept as low as possible. In order to achieve this, the interconnection subsystem used to support the interchange of messages must be fast enough to avoid becoming a bottleneck. Nowadays, the networks used in clusters are fast and have low latency. However, they can suffer from contention risks due to communication-intensive applications. As a consequence, the message latency is considerably increased, and the global system performance is degraded.

In addition, the scientific applications usually work with a large set of data that must be transferred among the processes and also stored in disk, provoking several hot spots in communications, but also in I/O operations.

Finally, the recent popularity of Chip Multiprocessors (CMP) increases the computational capability of clusters. Nevertheless, one of major limits of this kind of cluster is the I/O operations, because I/O requests initiated by multicore may saturate the I/O bus.

Therefore, this thesis targets reduction of overhead caused by communication and I/O subsystems applying two different strategies:

- **Reduction in the number of communications in collective I/O operations:** This thesis targets the reduction of the bottleneck in the I/O subsystem. Many applications use collective I/O operations to read/write data from/to disk. One of the most used is the *Two-Phase I/O* technique extended by Thakur and Choudhary in *ROMIO*. In this technique, many communications among the processes are performed, which could create a bottleneck. This bottleneck is still higher in commodity clusters, where commercial networks are usually installed, and in CMP clusters where the I/O bus is shared by the cores of a single node. Therefore, we propose improving locality in order to reduce the number of communications performed in *Two-Phase I/O*.

- **Reduction of transferred data volume:** This thesis attemps to reduce the cost of interchanged messages, reducing the data volume by using lossless compression among processes. Furthermore, we propose turning compression on and off and selecting at run-time the most appropriate compression algorithms depending on the characteristics of each message, network performance, and compression algorithms behavior.

If we reduce the number of communications in collective I/O and the volume of data transferred in communications, we hope to achieve improvement in the communications performance, producing an impact on the overall performance, and enhancing scalability of MPI applications.

1.4 Roadmap

This document is structured as follows:

- Chapter 2 introduces the basic concepts that are used throughout the thesis and overviews related work.
- Chapter 3 describes the new technique developed to reduce the number of communications during the I/O phase.
- Chapter 4 presents how the volume of data is reduced by using adaptive compression in run-time.
- Chapter 5 describes how we have integrated both techniques together in the same MPICH implementation.
- Chapter 6 reports the evaluations of the new techniques in MPICH.
- Chapter 7 contains a summary of the main conclusions and contributions.

Finally, the bibliography used in this thesis is included at the end of the document.

Chapter 2

State of the Art

This section presents the related work in the different fields that are covered in this Ph.D. thesis:

- Distributed Computing.
- Parallel Programming Paradigms.
- Parallel I/O Optimization Techniques.
- Compression Algorithms.
- Communication Optimization.

2.1 Distributed Computing

In recent years, parallel and distributed processing have been conjectured to be the most promising solution to the computing requirements of the future. Significant advances in parallel algorithms and architectures have demonstrated the potential for applying concurrent computation techniques to a wide variety of problems. The most representative distributed computing architectures are: Cluster, Grid and Cloud Computing.

Cluster and Grid Computing represent [YVCJ07] different approaches to solving performance problems. Although their technologies and infrastructure differ, their features and benefits complement each other. A cluster and a grid can run on the same network at the same time, and a cluster can even contribute resources to a grid.

Cloud Computing is also a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.

In the next Subsections we detail the main features of *Cluster, Grid and Cloud computing*.

2.1.1 Cluster Computing

The first inspiration for cluster computing was developed in the 1960s by IBM as an alternative to link large mainframes to provide a more cost effective form of commercial parallelism [Raj99]. A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource. A computer node can be a single or multiprocessor system (with memory, I/O facilities, and an operating system).

In 1994, Thomas Sterling and Don Becker, who worked for CEDIS (Center of Excellence in Space Data and Information Sciences), connected 16 systems with Intel 486 DX4 processors tied together with 10 Mbps channel bonded Ethernet (the processors were too fast for the standard 10 Mbps). That experiment grew into what is now called the Beowulf Project, a group of interested parties from the research and academic communities who work toward the development of related commodity off-the-shelf clusters. Beowulf class clusters are traditionally distinguished by low or no cost open source software for the operating system (usually Linux) and the message passing interface.

The typical architecture of a cluster computer is shown in Figure 2.1. The key components of a cluster include, multiple stand-alone computers (PCs, Workstations, or SMPs), an operating systems, a high performance interconnect, communication software, middleware, and applications.

The network interface hardware acts as a communication processor and is responsible for transmitting and receiving packets of data between cluster nodes via a network/switch.

The cluster nodes can work collectively, as an integrated computing resource, or they can operate as individual computers. The cluster middleware is responsible for offering an illusion of a unified system image and availability out of a collection in independent but interconnected computers.

Programming environments can offer portable and efficient tools for development of applications. They include message passing libraries, debuggers and profilers.

Clusters are classified [BB99] into many categories based on various factors as indicated below:

- **Application Target**: Computational science or mission-critical applications.

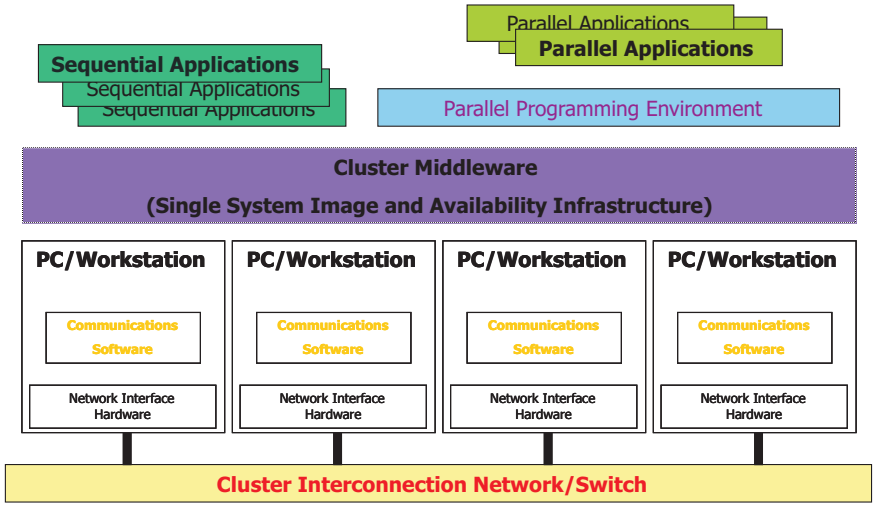


Figure 2.1: A Cluster Architecture.

  - High Performance (HP) Clusters: HPC clusters [Raj99] are designed to exploit the parallel processing power of multiple nodes. They are most commonly used to perform functions that require nodes to communicate as they perform their tasks, for instance, when calculation results from one node will affect future results from another.
  - High Availability (HA) Clusters: HA Clusters are designed to ensure constant access to service applications. The clusters are designed to maintain redundant nodes that can act as backup systems in the event of failure. Althought, the minimum number of nodes in an HA cluster is two (one active and one redundant) most HA clusters will use considerably more nodes. HA clusters aim to solve the problems that arise from mainframe failure in an enterprise. Rather than lose all access to IT systems, HA clusters ensure 24/7 access to computational power. This feature is especially important in enterprise applications, where data processing is usually time-sensitive.
  - Load balancing (LB) Clusters: LB Clusters [WSH06] are designed to load balance traffic, for example on high-traffic Web sites. Load balancing scales the performance of server-based programs, such as a Web server, by distributing client requests across multiple servers.

- **Node Ownership**: Owned by an individual or dedicated as a cluster node
  - Dedicated Clusters: They do not own a workstation. The resources are shared so that parallel computing can be performed across the entire cluster.
  - Nondedicated Clusters: They are individual workstations and applications are executed by stealing idle CPU cycles.

- **Node Hardware**: PC, WorkSation or SMP.

  - Clusters of PCs (CoPs)or PCs (PoPs)
  - Clusters of Workstation (COWs)
  - Clusters of SMPs (CLUMPS)

- **Node Operate System**: Linux, NT, Solaris, AIX, etc
  - Linux Clusters (e.g, Beowulf)
  - Solaris Clusters (e.g Berkeley NOW)
  - NT Clusters (e.g HPVM)
  - AIX Clusters (e.g IBM SP2)
  - Didigital VMS Clusters
  - HP-UX clusters
  - Microsoft Wolfpack clusters

- **Node Configuration**: Node architecture and type of OS are loaded with
  - Homogeneous Clusters: Compute nodes have similar architectures and run the same OSs.
  - Heterogeneous Clusters: Compute nodes have different architectures or run different OSs.

Computer clusters offer a number of benefits over mainframe computers, including:

- Reduced Cost: The price of off-the-shelf consumer desktops has plummeted in recent years, and this drop in price has corresponded with a vast increase in their processing power and performance. The average desktop PC today is many times more powerful than the first mainframe computers.

- Processing Power: The parallel processing power of a high-performance cluster can, in many cases, prove more cost effective than a mainframe with similar power. This reduced price per unit of power enables companies to obtain greater return of the investment (ROI) from their IT budget.

- Improved Network Technology: A driving force behind the development of computer clusters has been the vast improvement in technology related to networking, along with a reduction in the price of such technology.

- Computer clusters are typically connected via a single virtual local area network (VLAN), and the network treats each computer as a separate node. Information can be passed throughout these networks with very little lag, ensuring that data does not bottleneck between nodes.

- Scalability: Perhaps the greatest advantage of computer clusters is the scalability they offer. While mainframe computers have a fixed processing capacity, computer clusters can be easily expanded as requirements change by adding additional nodes to the network.

- Availability: When a mainframe computer fails, the entire system fails. However, if a node in a computer cluster fails, its operations can be simply transferred to another node within the cluster, ensuring that there is no interruption in service.

2.1.2 Grid Computing

In 1998 Foster and Kesselman [Kes98] defined the Grid as follows: *A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.*

In fact, one of the main ideas of the Grid, which also explains the origin of the word itself, was to make computational resources available, such as electricity. One remarkable fact of the electric power grid infrastructure is that when we plug an appliance into it we do not care where the generators are located and how they are wired.

The first generation grid systems involved solutions for sharing high performance computing resources [SHL+06]. The Information Wide Area Year (I-WAY) project [I-W95] is a representative first generation grid system in which the virtual environments, datasets, and computers resided at seventeen different US sites and were connected by ten networks. This project strongly influenced subsequent Grid computing activities. In fact one of the researchers who lead the project I-WAY was Ian Foster, who along with Carl Kesselman, published in 1997 a paper [FK96] that clearly links the Globus Toolkit, which is currently at the heart of many Grid projects, to Metacomputing.

Grid computing can be explained as [Dav04] applying the resources of many computers in a network to a single one at the same time. Usually, the problem is a scientific or technical problem that requires a large number of computer processing cycles or access to large amounts of data.

Grid computing requires the use of software that can divide and farm out pieces of a program to as many as several thousand computers. Globus Toolkit [FK99] is an open source toolkit for building computing grids, letting people share computing power, databases, and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit includes software services and libraries for resource monitoring, discovery, and management, plus security and file management. In addition to being a central part of science and engineering projects that total nearly a half-billion dollars internationally, the Globus Toolkit is a substrate on which leading IT companies are building significant commercial Grid products.

The toolkit includes software for security, information infrastructure, resource

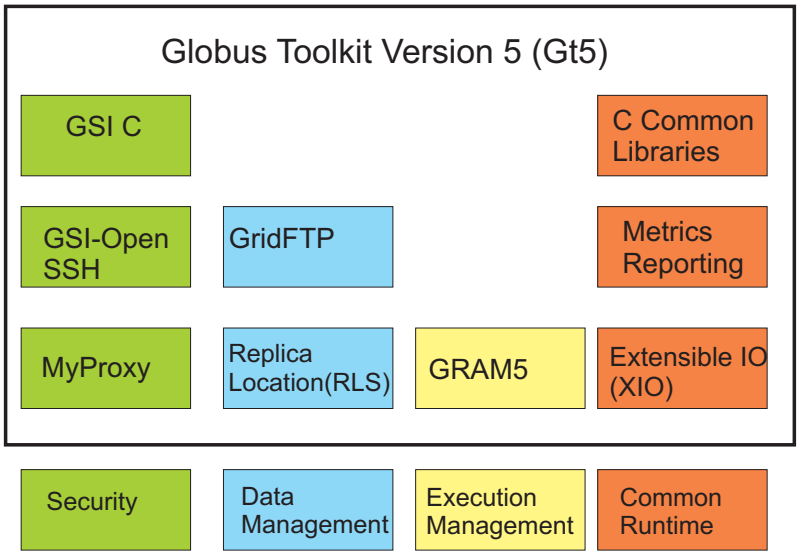


Figure 2.2: Globus Toolkit (GT5) Architecture.

management, data management, communication, fault detection, and portability. It is packaged as a set of components that can be used either independently or together to develop applications. Every organization has unique modes of operation, and collaboration among multiple organizations is hindered by incompatibility of resources such as data archives, computers, and networks.

The current Globus Toolkit version is 5. As we can observe in Figure 2.2, Globus architecture implies various complex system components, such as information infrastructure and resource management, data management, collaborative communication, fault detection and security.

There are many variations and types of Grids. Grid systems can be classified depending on their usage [KBI01]:

- Computational Grid: This is a system that has a higher aggregate capacity than any of its constituent machines. Depending on how this capacity is utilized, these systems can be further subdivided into:
  - Distributed SuperComputing: This kind of Grid, executes the application in parallel on multiple machines to reduce the completion time of a job. Typically, applications that require distributed supercomputing are for issues of great important challenge such as weather modelling and nuclear simulation.
  - High Throughput: Increases the completion rate of a stream of jobs arriving in real time. High Throughput grids are well suited for *parameter sweep* type applications such as Monte Carlo simulations.
- Data Grid: This is a system that provides an infrastructure for synthesizing new information from data repositories such as digital libraries or data

warehouses. The applications for these systems would be special purpose data mining that correlates information from multiple different high volume data sources.

- Service Grid: This is a system that provides services that are not provided by any single machine. This category is further subdivided based on the type of service they provide.
  - On Demand: This category dynamically aggregates different resources to provide new services.
  - Collaborative: This connects users and applications into collaborative workgroups, enabling real time interaction between humans and applications via a virtual workspace.
  - Multimedia: This provides an infrastructure for real time multimedia applications. It requires the support quality of service across multiple different machines whereas a multimedia application in a single dedicated machine can be deployed without QoS.

Grids are considered to provide multiple advantages to their participants. Despite the fact that the Grid is not limited in compute resources, the typical application of the Grid is currently still the execution of computational intensive tasks in high-performance computers. The resource providers are compute centers that share their processing powers for dedicated user communities. Besides the advantage of accessing locally unavailable resources, the Grid is also usually expected to utilize existing resources more efficiently. That is, machines are better utilized and users are getting a better quality of service, for instance, a shorter response time.

2.1.3 Cloud Computing

The latest paradigm emerging in distributed Computing is Cloud Computing [BYV08], which promises reliable services delivered through next-generation data centers that are built on compute and storage virtualization technologies. Consumers will be able to access applications and data from a "Cloud" anywhere in the world on demand. In other words, Cloud appears to be a single point of access for all the computing needs of consumers. The consumers are assured that the Cloud infrastructure is very robust and will always be available at any time.

A Cloud can be defined as a type of parallel and distributed system consisting of a collection of interconnected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers. At a glance, Clouds appear to be a combination of clusters and Grids. However, this is not the case. Clouds are clearly next-generation data centers with nodes "virtualized" through hypervisor technologies such as VMs, dynamically "provisioned" on demand as a personalized resource collection to meet a specific service-

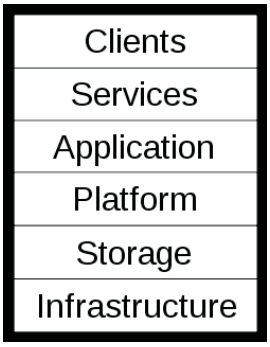


Figure 2.3: Cloud Architecture.

level agreement, which is established through a "negotiation" and accessible as a composable service via "Web 2.0" technologies.

There are several recognized layers in Cloud Computing. The vendors in these layers have very different service offerings and operating models. Some vendors concentrate on building and maintaining a huge data center, while others concentrate on building a user friendly and feature-rich application. The layers, from bottom to top, are shown in Figure 2.3: infrastructure, storage, platform, application, services, and client. As follows, we detail each one:

- Infrastructure: At the bottom is the infrastructure of the service. Cloud infrastructure services or "Infrastructure as a Service (IaaS)" deliver computer infrastructure, typically a platform virtualization environment as a service. Rather than purchasing servers, software, data center space or network equipment, clients instead buy those resources as a fully outsourced service. The service is typically billed on a utility computing basis and the amount of resources consumed (and therefore the cost) will typically reflect the level of activity. It is an evolution of virtual private server offerings.

- Storage: The storage layer is similar to a database, and pay per gigabyte per month. A storage layer is nothing new or special, except for the full stack of services. There are several possibilities for storage. Some are traditional relational databases, and some are proprietary solutions such as Google Bigtable or Amazon SimpleDB.

- Platform: Cloud platform services or "Platform as a Service (PaaS)" deliver a computing platform and/or solution stack as a service, often consuming cloud infrastructure and sustaining cloud applications. They facilitate deployment of applications without the cost and complexity of buying and managing the underlying hardware and software layers.

- Application: Cloud application services or "Software as a Service (SaaS)" deliver software as a service over the Internet, eliminating the need to install and run the application on the customer's own computers and simplifying maintenance and support.

- Services: The services layer consists of computer hardware and/or computer software products that are specifically designed for the delivery of cloud services, including multi-core processors, cloud-specific operating systems and combined offerings. The most prevalent example of this layer is Web services. Other examples include payments systems, such as Paypal, and mapping services, such as Google Maps and Yahoo Maps.

- Clients: At the top of the stack is the clients layer, which consists of computer hardware and/or computer software that relies on Cloud Computing for application delivery, or that is specifically designed for delivery of cloud services and that, in either case, is essentially useless without it. Examples include some computers, phones and other devices, operating systems and browsers. Clients are, for example, desktop users, and mobile users (Symbian, Android, iPhone).

This cloud model promotes availability and is composed of five essential characteristics and three deployment models [Eis09]:

Features:

- On-demand self-service: A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.

- Broad network access: Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs).

- Resource pooling: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines.

- Rapid elasticity: Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly be released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time.

- Measured Service: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service.

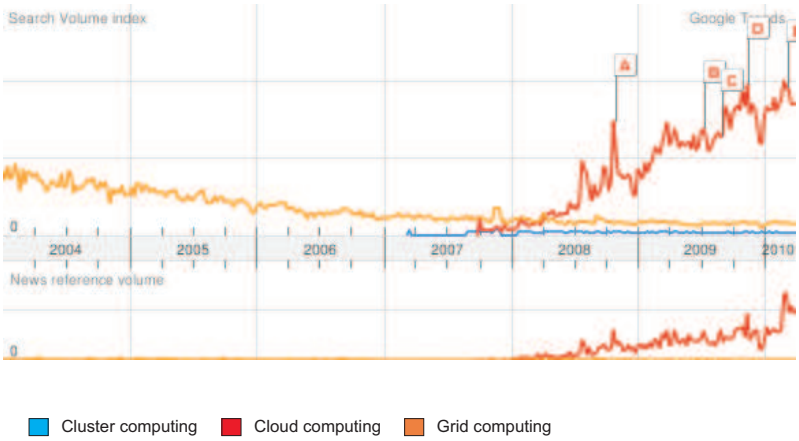


Figure 2.4: Google search trends for the last 6 years.

Deployment Models:

- Private cloud: The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.
- Community cloud: The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.
- Public cloud: The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.
- Hybrid cloud: The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).

The popularity of different distributed computing paradigms varies with time. The Web search popularity, as measured by the Google search trends during recent years, for the terms Cluster computing, Grid computing, and Cloud computing is shown in Figure 2.4. Spot points in Figure 2.4 indicate the release of news related to Cloud computing as follows:

- A: Microsoft to rent Web Cloud Computing space; Canada.com - Oct 28 2008.
- B: GSA Inches Closer To Cloud Computing 'Storefront'; InformationWeek - Jul 15 2009.
- C: Hosting.com Introduces vCloud Express - An On-Demand, Pay-As-You-Go Cloud Computing. Reuters - Sep 1 2009.

- D: Microsoft's Cloud Computing system is growing up; Philadelphia Inquirer - Nov 17 2009.
- E: Google looks to be Cloud Computing rainmaker for other online business services; Winnipeg Free Press - Mar 10 2010.
- F: Host Analytics Named to 2010 AlwaysOn OnDemand Top 100 List Recognizing the Leading SaaS and Cloud Computing Companies; MarketWatch - Apr 6 2010.

2.2 Parallel Programming Paradigms

Parallel computing philosophy lies in fact that the computing task itself can be divided into smaller tasks in order to obtain better performance by executing them in parallel. This can be acomplished by using different platforms (multicore, clusters, grid, etc.)

The approach differs in the type of hardware architecture that will be used. Memory architecture in a parallel computer is either shared memory or distributed memory. In the first one, the memory is shared by all processing elements in a single address space. In the second one, each processing element has its own local address space. Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well.

There are several alternatives for programming for shared memory:

- Using a new programming language like Cilk [MIT10].
- Modifying an existing sequential language like linda [Wel05] or OpenMP [Ope05].
- Using library routines like Intel TBB (Intel Threading Building Blocks for C++) [Rei07].
- Using a sequential programming language and asking a parallelizing compiler to convert it into a parallel executable code like some Fortran compilers with automatic parallelization.
- Threads (POSIX threads, Java, ISO threads, etc.) with some concurrency control mechanism for controlling access to shared memory like software transactional memory [HM93].

Also there are several alternatives for programming with distributed memory, mainly:

- MPI (Message Passing Interface) can be used with distributed memory (as well as with the shared memory model).
- PVM (Parallel Virtual Machine) [KHQ+94].

Today systems (based on many-core CPU) make use of multi-level parallel programming (e.g. MPI+OpenMP), a hot topic for developers.

The following Subsections describe each of the models mentioned above, and also discuss some of their actual implementations.

2.2.1 Threads Model

The thread model [Ead06] is a way for a program to split itself into two or more concurrent tasks. These tasks can be run on a single processor in a time-shared mode, or on separate processors (e.g. the two cores on a dual-core processor can each run threads). The term thread comes from *thread of execution* and is similar to how a fabric (computer program) can be pulled apart into threads (concurrent parts). Threads are different from individual processes (or independent programs) because they inherit much of the state information and memory from the parent process.

In Linux and Unix systems, threads are often implemented using a POSIX (Portable Operating System Interface) [IEE90] Thread Library (pthreads). There are several other thread models (Windows threads) from which the programmer can choose; However, using a standards based implementation, such as POSIX, is highly recommended. As a low level library, pthreads can be easily included in almost all programming applications.

Threads provide the ability to share memory and offer very fine-grained synchronization with other sibling threads. These low level features can provide very fast and flexible approaches to parallel execution. Software coding at the thread level is not without its challenges. Threaded applications require attention to detail and considerable amounts of extra code to be added to the application. Finally, threaded applications are ideal for multi-core designs because the processors share local memory.

Because native thread programming can be cumbersome, a higher level abstraction has been developed called OpenMP[Ope05]. As with all higher level approaches, there is a sacrifice of flexibility for the ease of coding. At its core, OpenMP uses threads, but the details are hidden from the programmer. Specifications exist for C/C++ and FORTRAN, several of the most commonly used languages for writing parallel programs. The standard provides a specification of compiler directives, library routines, and environment variables that control the parallelization and runtime characteristics of a program. Since it is a standard which is enjoying increasing levels of implementation, code written with OpenMP is portable to other shared-memory multiprocessors. The compiler directives defined by OpenMP tell a compiler which regions of code should be parallelized and define specific options for parallelization. In addition, some precompiler tools exist which can automatically convert serial programs into parallel programs by inserting compiler directives into appropriate places, making the parallelization of a program even easier. One example is the now discontinued product from Kuck and Associates (now owned by KAI Software), Visual KAP [Vis01] for OpenMP.

OpenMP is based on a thread paradigm [Qua02]. A running program, referred to as a process, is allocated its own memory space by the operating system when the program is loaded into memory. Within a process, multiple threads may exist. A thread is an active execution sequence of instructions within a process. Threads within a process share the same memory space and can access the same variables. They have the advantage of allowing a process to perform multiple tasks seemingly simultaneously. For example, a web browser may have a thread that requests and receives web pages, another thread to render web pages for display on the screen, and yet another thread to "listen" for user input and respond appropriately. Without threads, the web browser might be required to block while waiting for a web page to download, preventing a user from doing things such as accessing a pull-down menu.

The thread paradigm is a logical choice for a shared-memory multiprocessor or multicore system. The concept is based on the fork-join model of parallel computation. A master thread runs serially until it encounters a directive to fork off new threads. These threads can then be distributed and executed on different processors, reducing execution time since more processor cycles are available per time unit. Results of each threads execution can then be combined. A user can set the number of threads created for a parallel region by setting the environment variable OMP_NUM_THREADS, or the programmer can set it using the library call omp_set_num_threads.

2.2.2 Message Passing Model

Programming models are generally categorized by the way memory is used [Com07]. In the shared memory model each process accesses a shared address space, while in the message passing model an application runs as a collection of autonomous processes, each with its own local memory. In the message passing model, processes communicate with other processes by sending and receiving messages. When data are passed in a message, sending and receiving processes must work to transfer the data from the local memory of one to the local memory of the other.

Message passing is widely used on parallel computers with distributed memory, and on clusters of servers. The advantages of using message passing include:

- Portability: Message passing is implemented on most parallel platforms.
- Universality: The model makes minimal assumptions about underlying parallel hardware. Message-passing libraries exist on computers linked by networks and on shared and distributed memory multiprocessors.
- Simplicity: The model supports explicit control of memory references for easier debugging.

However, creating message-passing applications may require more effort than letting a parallelizing compiler produce parallel applications. In 1994, representatives from the computer industry, government labs, and academe developed a standard

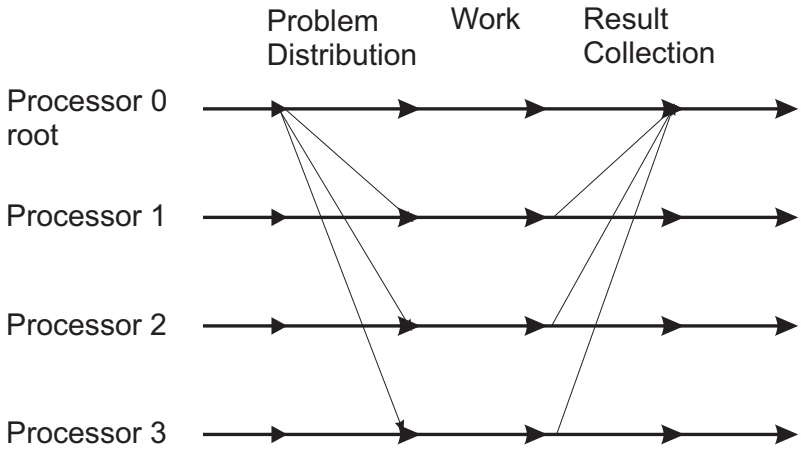


Figure 2.5: MPI execution model.

specification for interfaces to a library of message-passing routines. This standard is known as MPI [Mes94] (MPI: A Message-Passing Interface Standard).

Implementation of the standard is usually left up to the designers of the systems on which MPI runs, but a public domain implementation, MPICH, is available. MPI is a set of library routines for C/C++ and FORTRAN. Like OpenMP, MPI is a standard interface, so code written for one system can easily be ported to another system with those libraries.

The execution model of a program written with MPI is quite different from one written with OpenMP. When an MPI program starts, the program spawns into the number of processes as specified by the user. Each process runs and communicates with other instances of the program, possibly running on the same processor or different processors. The greatest computational speedup will occur when processes are distributed among processors. Basic communication consists of sending and receiving data from one process to another, unlike OpenMP's thread communication via shared variables. This communication takes place over a high-speed network which connects the processors in the distributed-memory system.

A data packet sent with MPI requires several pieces of information: the sending process, the receiving process, the starting address in memory of the data to be sent, the number of data items being sent, a message identifier, and the group of processes that can receive the message. All these items are able to be sent by the programmer. For example, one can define a group of processes, then send a message only to that group.

Some collective communication routines do not require all items. For example, a routine which allows one process to communicate with all other processes in a group when called by each of those processes would not require the specification of a receiving process since every process in the group should be a receiver.

In the simplest MPI programs, a master process sends off work to worker processes. Those processes receive the data, perform tasks on it, and send the results back to the master process which combines the results. More complex coordination

schemes are possible with MPI, but they introduce challenges which will be discussed shortly. Note that the other processes run continuously from the launch of the program, a difference from the OpenMP fork-join model. Figure 2.5 shows the execution model of a basic MPI program.

Currently, there are many implementations of MPI, with both free available and vendor-supplied implementations, but few implementations provide a fully thread-safe semantic. Some of these systems are as follows:

- LAM [GDV94] is available from the Ohio Supercomputer Center and runs on heterogeneous networks of Sun, DEC, SGI, IBM, and HP workstations.
- CHIMP-MPI [RBMA94] is available from the Edinburgh Parallel Computing Center and runs on Sun, SGI, DEC, IBM, and HP workstations, the Meiko Computing Surface machines.
- MPICH [GLDS96] is an implementation developed at Argonne National Laboratory and the Mississippi State University. This implementation is available for several platforms, Sun, SGI, RS6000, HP, DEC and Alpha workstations and multicomputers as the IBM SP2, Meiko CS-2 and Ncube.
- FT-MPI (Fault Tolerant MPI) is a full 1.2 MPI specification implementation that provides process level fault tolerance at the MPI API level. FT-MPI is built upon the fault tolerant harness runtime system. This implementation is developed at University of Tennessee.
- LA-MPI is an implementation of the Message Passing Interface (MPI) motivated by a growing need for fault tolerance at the software level in large high-performance computing (HPC) systems. This implementation has two primary goals: network fault tolerance and high performance. LA-MPI is developed by the Application Communications and Performance Research Team of the Advanced Computing Laboratory at LANL (Los Alamos National Laboratory).
- OpenMPI [GFB+04] is a MPI implementation which combines technologies and resources from several other projects (FT-MPI, LA-MPI, and LAM/MPI). It is used by many TOP500 supercomputers including Roadrunner, which was the world's fastest supercomputer from June 2008 to November 2009.
- Unify [VSRC95], available from Mississippi State University, layers MPI on a version of PVM that has been modified to support contexts and static groups. Unify allows MPI and PVM calls in the same program.
- MPI-LITE [P. 97] provides a portable kernel for thread creation, termination, and scheduling. This implementation can be used in multithread applications. However, this model is very strict due to the fact that it uses user-level threads; Each thread executes a copy of the given MPI program, and the total number of threads in the program is specified as inputs to the MPI-LITE program.

This implementation does not support dynamic creation and termination of threads

- MiMPI [GCC99] was a prototype of a multithread implementation of MPI with thread-safe semantics that adds run-time compression of messages sent among nodes.

2.2.3 Data Parallel Model

The term data parallelism refers to the concurrency that is obtained when the same operation is applied to some or all elements of a data ensemble. A data-parallel program is a sequence of such operations. A parallel algorithm is obtained from a data-parallel program by applying domain decomposition techniques to the data structures operated on. Operations are then partitioned, often according to the *owner computes* rule, in which the processor that owns a value is responsible for updating that value. Typically, the programmer is responsible for specifying the domain decomposition, but the compiler partitions the computation automatically.

A currently popular approach to portable data-parallel computing is based on the Fortran90 model, which extends the scalar arithmetic of Fortran77, and is now extended as High Performance Fortran model [KKZ07].

The Fortran90 based data-parallel model allows us to treat massively parallel machines as superfast mathematical co-processors/accelerators for matrix operations. The details of the parallel hardware architecture and even its existence are transparent to the Fortran programmer. Good programming practice is simply to minimize explicit loops and index manipulations and to maximize the use of matrices and index-free matrix arithmetic, optionally supported by the compiler directives to optimize data decompositions. The resultant product is a metaproblem programming system having as its core, for synchronous and loosely synchronous problems, an interpreter of High Performance Fortran.

2.3 Parallel I/O Optimization Techniques

As depicted in [Flo08], the performance of applications accessing large data sets is often limited by the speed of the I/O subsystems. Studies of I/O intensive parallel scientific applications [NKP+96b, SR98] have shown that an important performance penalty stems from the mismatch of the file striping (physical parallelism) and the access patterns (logical parallelism). In this context, it is important how the software layers between applications and disk, namely I/O libraries like MPI-IO and file systems, use the system parallelism. This work targets the optimization of the MPI-IO interface inside ROMIO, the most popular MPI-IO distribution.

The are several I/O techniques to improve the performance of I/O subsystems. In this thesis we shall focus on collective I/O techniques. Collective I/O addresses this problem by merging small individual requests into larger global requests in order

to optimize the network and disk performance. Depending on the place where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at the I/O nodes, the method is called *disk-directed I/O* [KCJ+95]. If the merging occurs at intermediary nodes, or at compute nodes the method is called two-phase I/O [Bor97].

In the next subsections we will briefly describe these two collective I/O techniques.

2.3.1 Two-Phase I/O

Two-phase I/O performs two steps as illustrated in Figure 2.6 for the collective read case. Compute node 1 issues a read for the dark grey bytes, while compute node 2 attempts to read the light grey bytes. In the access phase, the compute nodes divide the access interval into equal parts after a negotiation (1) and each reads contiguously its share from the file system into a local collective buffer (2 and 3). In the shuffle phase (4), the compute nodes exchange the data according to the requested access pattern. The access phase is always fast, as only contiguous requests are sent to the file system. The scatter-gather operations take place at the compute node, whereas the data travels twice through the network.

The Two-Phase technique was extended by Thakur and Choudhary [RC95] by balancing the load on the processors that perform I/O and by reducing the number of requests by data sieving. Extended two-phase I/O is an optimization of ROMIO [RGL99], an implementation of MPI-IO interface.

2.3.2 Disk-directed I/O

Figure 2.7 shows a collective read example for disk-directed I/O [Dav94]. The compute nodes send the requests directly to the I/O nodes (1). I/O nodes merge and sort the requests (2) and send them to disk (3). The data is read from the disk (4), gathered in network buffers (5) and sent to compute nodes (6).

A method related to disk-directed I/O is server-directed I/O as implemented in the Panda library [KCJ+95]. The compute nodes send to a master I/O server a short high-level description of the in-memory and on-disk distributions. The master server then provides all the other I/O servers running on I/O nodes with the distribution information and each server independently plans how it will request or send its assigned disk data to or from the relevant clients. The main difference between the two methods is that server-directed I/O operates at a higher level of abstraction (files in the local file system on the I/O nodes) than disk-directed I/O (which handles disk blocks).

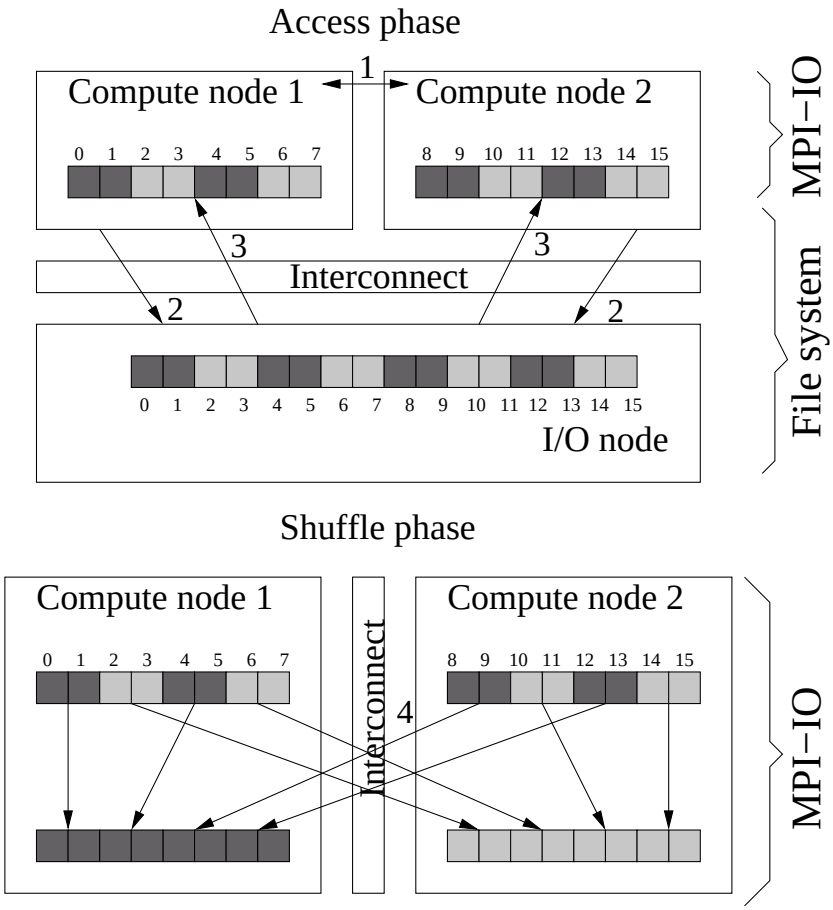


Figure 2.6: Two-Phase I/O read example.

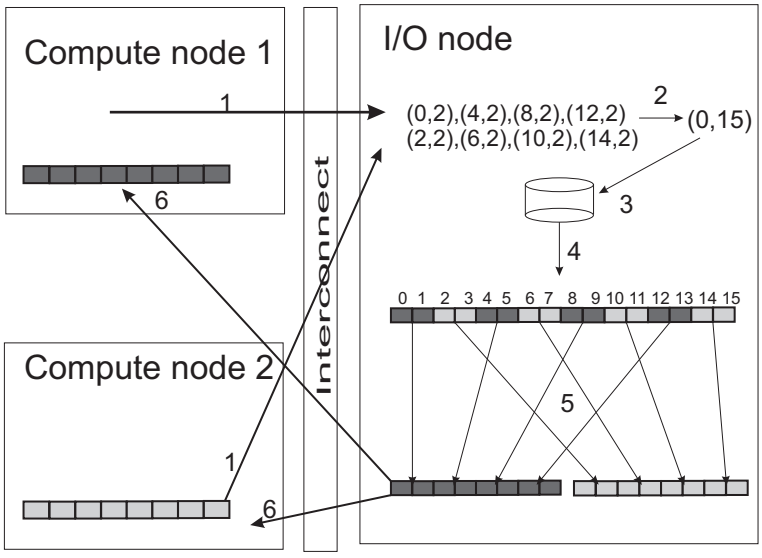


Figure 2.7: Disk-directed I/O read example

.

2.4 Compression Algorithms

Compression is the science of reducing the amount of data used to convey information [Sym00].

The process of transforming information from one representation to another, a smaller representation from which the original, or a close approximation to it, can be recovered. The compression and decompression processes are often referred to as encoding and decoding. Data compression has important applications in the areas of data storage and data transmission. Besides compression savings, other parameters of concern include encoding and decoding speeds and workspace requirements, the ability to access and decode partial files, and error generation and propagation.

Compression is useful because it helps to reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth. On the downside, compressed data must be decompressed to be used, and this extra processing may be detrimental to some applications [AMI09]. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed (the option of decompressing the video in full before watching it may be inconvenient, and requires storage space for the decompressed video). The design of data compression schemes therefore involves trade-offs among various factors, including the degree of compression, the amount of distortion introduced in the image, and the computational resources required to compress and uncompress the data.

The data compression process is said to be lossless if the recovered data are assured to be identical to the source, Figure 2.8 as shows. Otherwise the compression process is said to be lossy. Lossless compression techniques are requisite for applications involving textual data. Other applications, such as those involving voice and image data, may be sufficiently flexible to allow controlled degradation in the data.

One of our targets in this thesis is to improve the performance of scientific applications in clusters by applying compression in MPI (by compressing the transferred messages among the processes). This implies some important requirements:

- The compression algorithm must be a lossless algorithm in order to preserve the information.
- The compressor must produce the smallest overhead possible in terms of execution time and memory requirements. Because of that, our priority is speed, instead of the compression ratio.

In the literature, there are many studies and evaluations of data compression techniques that have been evaluated with different datasets. Nowadays, the best known benchmark to evaluate lossless compression methods is the Canterbury Corpus [AB97]. In the associated web site, visitors can find results for compression

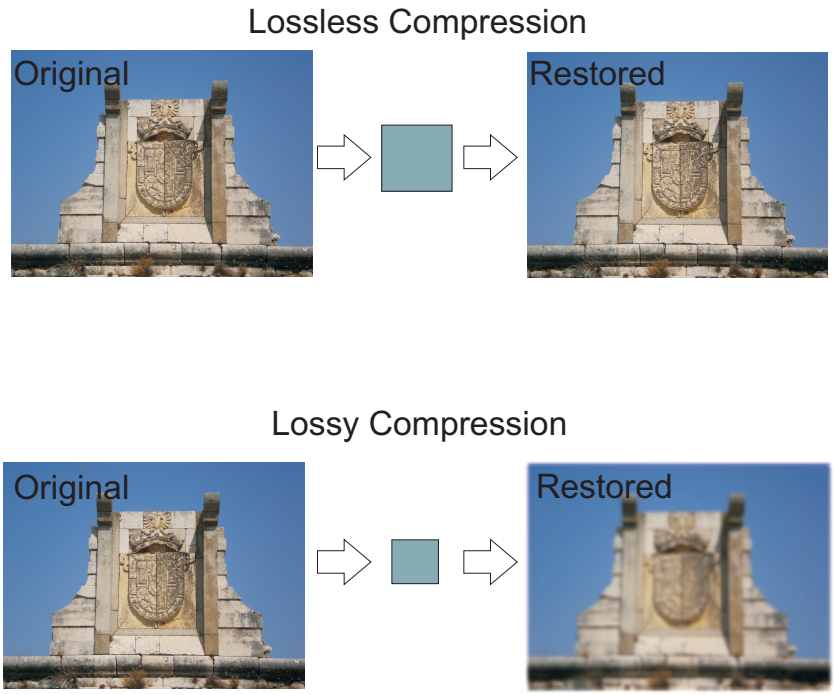


Figure 2.8: Example of Lossles and Lossy Compression.

ratio, compression time, and decompression time of several different algorithms. Unfortunately, the Canterbury Corpus is a general purpose benchmark, and many of our compressors are not present in its results. For these reasons, this work also introduces a novelty benchmarking for selecting compressors in high performance environments.

In this thesis we have tested many compressors. Based on those tests, we have preselected a group of compressors fitting our requirements to evaluate their suitability for their inclusion in our MPI platform. The compressors selected have been previously used in some high performance environments (such as PACX-MPI, MiMPI, etc.) or in real-time systems where the time needed for compression/decompression is critical. Compressors selected are: RLE [Zig89], Huffman [Knu85], Rice [SC00], FPC [BR09] and LZO [Obe05]. Table 2.1 shows the main characteristics of each one of them. As may be seen, each compressor fits a specific data type better, which will allow us to adapt compression to the type of data of each message.

The following Subsections describe each of the compression algorithms mentioned in the Table 2.1.

2.4.1 LZO

LZO (Lempel-Ziv-Oberhumer) [Mar02] is a data compression library which is suitable for data decompression in real-time. LZO offers fairly fast compression and "extremely" fast decompression. In addition, it includes slower compression levels achieving a quite competitive compression ratio while still decompressing at this

very high speed.

LZO is written in ANSI C. Both the source code and the compressed data format are designed to be portable across platforms. LZO implements a number of algorithms with the following features:

- Decompression is simple and fast.
- It requires little memory for decompression.
- Compression is fairly fast.
- It requires 64 KB of memory for compression.
- It allows you to dial up extra compression at a speed cost in the compressor. The speed of the decompressor is not reduced.
- Includes compression levels for generating pre-compressed data which achieve a quite competitive compression ratio.
- There is also a compression level which needs only 8 KB for compression.
- The algorithm is thread safe.
- The algorithm is lossless.

Name	Characteristics	Recommended data type
LZO	Blocks compression algorithm with very low compression and decompression time	Binary (image, audio, etc.)
RLE	Based on sequences of the same byte	Text
Huffman	Assigns short codes to frequent sequence of bytes. It works at byte level	Text and binary
Rice	Entropy-based encoding technique	Binary (image, audio, etc.)
FPC	Based on the prediction of the next value of data sequence	Doubles

Table 2.1: Compressors preselected.

2.4.2 RLE

Run Length Encoding (RLE) [Gol] is a simple and popular data compression algorithm. It is based on the idea of replacing a long sequence of the same symbol by a shorter sequence and it is a good introduction for newcomers to the data compression field.

The sequence of length l of a repeated symbol s is replaced by a shorter sequence, usually containing one or more symbols of s, a length information and sometimes an escape symbol.

For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. Let us take a hypothetical single scan line, with B representing a black pixel and W representing white:

WWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWW

If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:

10W1B11W3B20W1B12W

Interpret this as ten W's, one B, eleven W's, three B's, etc.

The RLE algorithm differs from the other mainly in three points: the threshold t, the marking of the start of a run, and the coding of the length information. If l is smaller than t, the run stays unchanged, and if l is greater than or equal to t, the run is replaced. The start of a run can be indicated by a threshold run or an escape symbol c. If a threshold run is used the start is characterized by a small sequence of s, which has a length greater than or equal to t. If an escape symbol c indicates the start of a run, s is normally put behind c in order to characterize the run symbol. c must not be an element of the alphabet or occurrences of c have to be coded in a way that they are not mixed up with the start of a run.

Thus, RLE compression makes no sense except for data with many consecutive repeated elements. This method, however, has the advantage of not being very difficult to implement.

2.4.3 Huffman

The Huffman compression algorithm [Knu85] is named after its inventor, David Huffman. Huffman compression is a lossless compression algorithm that is ideal for compressing text or program files.

Huffman compression belongs to a family of algorithms with a variable codeword length. That means that individual symbols (characters in a text file for instance) are replaced by bit sequences that have a distinct length. So symbols that occur a

lot in a file are given a short sequence while others that are used seldom get a longer bit sequence.

A practical example will show the principle: Suppose one wants to compress the following piece of data: "ACDABA".

Since these are 6 characters, this text is 6 bytes or 48 bits long. With Huffman encoding, the file is searched for the most frequently appearing symbols (in this case the character "A" occurs 3 times) and then a tree is built that replaces the symbols by shorter bit sequences. In this particular case, the algorithm would use the following substitution table: A=0, B=10, C=110, D=111. If these code words are used to compress the file, the compressed data look like this: 01101110100

This means that 11 bits are used instead of 48, a compression ratio of 4 to 1 for this particular file.

Huffman encoding can be further optimized in two different ways:

- Adaptive Huffman code dynamically changes the code words according to the change of probabilities of the symbols.

- Extended Huffman compression can encode groups of symbols rather than single symbols.

This compression algorithm is mainly efficient in compressing text or program files. Images like they are often used in prepress are better handled by other compression algorithms.

2.4.4 Rice

The Rice compression algorithm is a lossless compression algorithm [SC00]. It was developed in part by Robert Rice, and is a special case of Golomb coding, which comprised concatenation of unary and binary representations of the same symbol, based in part on a coefficient that could be optimized for the expected distribution of data. By restricting these coefficients to powers of 2, Rice coding greatly simplifies the decoding process, resulting in a coding system that can be more efficiently employed in electronic or computerized systems.

The coding process itself takes advantage of the fact that changes from element to element of certain types of data such as video image data is typically very small. By encoding these small but frequently occurring changes using code symbols that are small relative to the original data symbol size, lossless compression of the data can be achieved. Because Rice compression algorithms are particularly well suited to compression of image data, the algorithm has been implemented in many environments where lossless compression of images is needed, such as in satellite communication of image data.

2.4.5 FPC

FPC is a lossless, single-pass, linear-time compression algorithm [BR09]. FPC targets streams of double-precision floating-point data with unknown internal structure, such as the data seen by the network or a storage device in scientific and high-performance computing systems. If the internal structure is known, for example, a matrix or a linearized tree, then this extra information could be exploited to improve the compression ratio [LI06]. FPC delivers a good average compression ratio on hard-to-compress numeric data. Moreover, it employs a simple algorithm that is very fast and easy to implement with integer operations. We found FPC to compress and decompress 2 to 300 times faster than the special-purpose floating-point compressors DFCM, FSD and PLMI (described in [BR07]) and the general-purpose compressors BZIP2 and GZIP [lG10].

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value, applying the xor operation to the true value with the predicted value, and leading-zero compressing the result. As illustrated in Figure 2.9, it uses variants of an *fcm* [SS97] and a *dfcm* [GVdB01] value predictor to predict the doubles. Both predictors are effectively hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bits with the true value, is xored with the true value. The xor operation turns identical bits into zeros. Hence, if the predicted and the true value are close, the xor result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a three-bit value, and concatenates it with a single bit that specifies which of the two predictions was used. The resulting four-bit code and the nonzero residual bytes are written to the output. The latter are emitted verbatim without any encoding.

FPC outputs the compressed data in blocks. Each block starts with a header that specifies how many doubles the block encodes and how long it is (in bytes). The header is followed by the stream of four-bit codes, which in turn is followed by the stream of residual bytes. To maintain byte granularity, which is more efficient than bit granularity, a pair of doubles is always processed together and the corresponding two four-bit codes are packed into a byte. In case an odd number of doubles needs to be compressed, a spurious double is encoded at the end. This spurious value is later eliminated using the count information from the header.

Decompression works as follows. It starts by reading the current four-bit code, decoding the three-bit field, reading the specified number of residual bytes, and zero-extending them to a full 64-bit number. Based on the one-bit field, this number is xored with either the 64-bit fcm or dfcm prediction to recreate the original double. This lossless reconstruction is possible because xor is reversible.

For performance reasons, FPC interprets all doubles as 64-bit integers and uses only integer arithmetic. Since there can be between zero and eight leading zero bytes, for example, nine possibilities, not all of them can be encoded with a three-bit value.

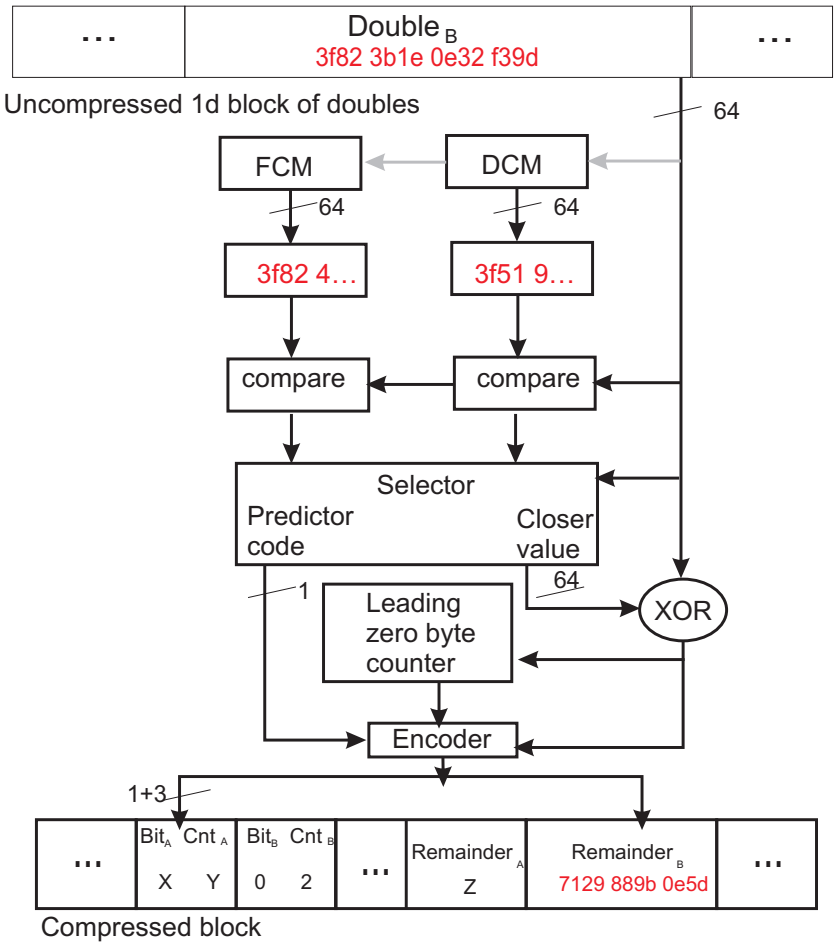


Figure 2.9: FPC compression algorithm overview.

FPC does not support a leading zero count of four because it occurs only rarely. Consequently, all xor results with four leading zero bytes are treated like values with only three leading zero bytes and the fourth zero byte is emitted as part of the residual.

Before compression and decompression, both predictor tables are initialized with zeros. After each prediction, they are updated with the true double value to ensure that they generate the same sequence of predictions during compression as they do during decompression.

2.5 Communication Optimizations

As follows, we present the solutions that have been presented to reduce the impact of communications in environments clusters. Therefore, we classify the solutions in:

- Reduction of transferred data volume.
- Reduction of number of communications.

Now, we explain briefly the most relevant works of each areas.

2.5.1 Reduction of transferred data volume

The techniques presented as follows focus on reducing application execution time in MPI based parallel application by using compression in order to reduce the transferred data volume. The use of compression within MPI is not new, although it has been particularized for very few special cases. Major examples of compression added into MPI tools are cMPI, PACX-MPI, COMPASSION and MiMPI.

PACX-MPI (PArallel Computer eXtension to MPI) [BTR03, RK05] is an ongoing project of the HLRS, Stuttgart. It enables an MPI application to run on a meta-computer consisting of several, possibly heterogeneous machines, each of which may itself be massively parallel. Compression is used for TCP message exchange among different systems in order to increase the bandwidth, but the same compression algorithm is always employed, and compression is not used for messages within the same system.

cMPI [RKB06, KBS04a] has some goals similar to PACX-MPI: to enhance the performance of inter-cluster communication with a software-based data compression layer. It use lossless compression schemes for compressing large messages of float values to improve large message latency and hence application performance. Compression is added to all communication, so it does not have flexibility to configure when and how to use it. They use a value prediction scheme which computes the difference between the actual and predicted data values and encodes the difference using the leading zero count (LZC)[KBS04a] method to compress the messages.

The work presented in [KNTG08] focuses on reducing communication time in message passing based parallel applications using compression. It evaluates the performance of both lossless and lossy compression schemes in the CAM application.

COMPASSION [CNP+98] is a parallel I/O runtime system including chunking and compression for irregular applications. The LZO algorithm is used for fast compression and decompression, but again it is only used for (and focused on) the I/O part of irregular applications.

MiMPI [GCC99] was a prototype of a multithreaded implementation of MPI with thread-safe semantics that adds run-time compression of messages sent among nodes. Although the compression algorithm can be changed (providing more flexibility), the use of compression is global for all processes of a MPI application. That is to processes say: all the MPI processes have to use the same compression technique at once. The compression is used for messages larger than a given size, and not can be tailored per message (based on datatype, source rank, destination rank, etc.).

2.5.2 Reduction of number of communications.

Currently, to reduce the number of communications, the new techniques are based on exploiting the shared memory of cluster Multicore. The main advantage of using

shared memory is to reduce some communications that otherwise would be performed by using the network.

Work explained in [HC06] studies several methods to improve the performance of MPI-IO, especially for non-contiguous accesses by using shared memory.

In addition, we want to highlight the new collective I/O technique for clusters MultiCore called *Asymmetric Computation* explained in the article [OCH+08]. In a cluster, I/O requests initiated by multiple cores may saturate the I/O bus, and furthermore increase the latency by issuing multiple non-contiguous disk accesses. In this paper, they propose an asymmetric collective I/O for multicore processors to improve multiple non-contiguous accesses. In this work, one core in each multicore processor is designated as the coordinator, and others serve as computing cores. The coordinator is responsible for aggregating I/O operations from computing cores and submitting a contiguous request. The coordinator allocates contiguous memory buffers on behalf of other cores to avoid redundant data copies.

Chapter 3

Data Locality Aware Strategy for Two-Phase Collective I/O

3.1 Introduction

As explained in Section 1.3, this Ph.D. thesis has as its main objective improving the scalability and performance of MPI-based applications executed in clusters. To reach this objective, in this chapter we have focused on reducing the bottleneck in the I/O subsystem.

When examining the structure of the scientific applications that run on parallel machines, we observe that their I/O needs to increase tremendously with each generation of software. In general, parallel applications work with very large data sets which, in most cases, do not fit in memory and have to be kept in disk. The input and output data files have a large size and have to be accessed very fast. These large applications also perform checkpointing operations that require large disk space resources and have to be completed as quick as possible (in order to not degrade the whole application performance). Parallel file systems such as GPFS [SH02], PVFS [LIR99] and Lustre [CFS02] offer scalable solutions for concurrent and efficient storage. These parallel file systems are accessed by the parallel applications through interfaces such as *POSIX* or *MPI-IO*. This thesis targets the optimization of MPI-IO interface inside ROMIO, the most popular MPI-IO distribution.

Many parallel applications consist of alternating compute and I/O phases. During the I/O phase, the processes frequently access a common data set by issuing a large number of small non-contiguous I/O requests [NKP+96c, SR98]. Usually these requests originate an important performance slowdown of the I/O subsystem. Collective I/O addresses this problem by merging small individual requests into larger global requests in order to optimize the network and disk performance.

One of the most used Collective I/O technique is *Two-Phase I/O* extended by Thakur and Choudhary in *ROMIO*. *Two-Phase I/O* takes place in two phases: a redistributed data exchange and an I/O phase. In the first phase, by means of communication, small file requests are grouped into larger ones. In the the second phase, contiguous transfers are performed to or from the file system. Before that, *Two-Phase I/O* divides the file into equal contiguous parts (called File Domains (FD)), and assigns each FD to a configurable number of compute nodes, called aggregators. Each aggregator is responsible for aggregating all the data which it maps inside its assigned FD and for transfering the FD to or from the file system.

In the default implementation of *Two-Phase I/O* the assignment of each aggregator (aggregator pattern) is fixed, independent of distribution of data over the processes. This fixed aggregator pattern might create a I/O bottleneck, as a consequence of the multiple requests performed by aggregators to collect all data assigned to their FD. This bottleneck is still higher in commodity clusters, where commercial networks are usually installed, and in CMP clusters where the I/O bus is shared among the cores of a single node.

Based on that, we propose replacing the rigid assignment of aggregators over the processes in order to reduce the number of communications performed in *Two-Phase I/O*. The result is a new dynamic I/O aggregator pattern based on local data that each process stores. In addition, our proposal can be used for any kind of application that uses *Two-Phase I/O*, in transparent way for the user.

3.2 Internal Structure of Two-Phase I/O

This chapter is focused on improving the *Two-Phase I/O* technique in order to enhance the scalability and performance of applications. Therefore, we consider it necessary to explain in detail the internal structure of this technique.

As was explained earlier, *Two-Phase* consists of two interleaved phases: a redistributed data exchange phase and the I/O phase. The *Two-Phase I/O* strategy first calculates the aggregate access file region and then partitions it among the I/O aggregators into approximately equal-sized contiguous segments, called file domains (FD). This partitioning strategy is referred to as the partitioning method. The I/O aggregators are a subset of the processes that act as I/O proxies for the rest of the processes.

In the data redistribution phase, all processes exchange data with the aggregators based on the calculated FDs. In the I/O phase, aggregators access the shared file within the assigned FDs transferring the data to it. The parallelism comes from the aggregators performing their writes/reads concurrently. This is more efficient because it is significantly more expensive to write to the file system than to perform inter-process communications. As follows, we enumerated the stages of *Two-Phase* previously explained:

- Previous calculations:
  - *Offsets and lengths calculation (st1)*: In this stage the lists of offsets and lengths of the file are calculated.
  - *Offsets and lengths communication (st2)*: Each process communicates its start and end offsets to the other processes. In this way all the processes have global information about the involved file interval.
  - *File domain calculation (st3)*: The I/O workload is divided among processes. This is done by dividing the file into file domains (FDs). In this way, in the following stages, each aggregator collects and transfers the data associated to its FD to the file.
- Redistributed Data Exchange Phase:
  - *Access request calculation (st4)*: This calculates the access requests for the file domains of remote aggregator.
  - *Metadata transfer (st5)*: It transfers the lists of offsets and lengths.
  - *Buffer writing (st6)*: The data are sent to the appropriate aggregator.
- I/O Phase:
  - *File writing (st7)*: The data are written to the file.

The buffer and file writing stages *(st6* and *st7)* are repeated as many times as the following calculus indicates: the size of the file portion of each aggregator is divided by the size of the *Two-Phase I/O* buffer (4 MB by default).

Note, that aggregators are defined at the time the file is opened (in MPI_File_ ADIO_Open function), and not during the execution of *Two-Phase I/O*. Therefore, when the *Two-Phase I/O* begins, it accesses the aggregator pattern. A collection of MPI hints can be used to tune what processes become aggregators for a given file. However, by default the aggregator pattern uses one process per node as aggregator and the assignment of aggregators is independent of the distribution of data over the processes. This means that, if there are four processes in four different nodes, the four processes will become aggregators. If the four processes are instead located in two nodes, only two of them will become aggregators.

Process 0 is responsible for determining which processes are aggregators. A previous call to *ADIOI_cb_gather_name_array* collects the processor names from all hosts into an array that is cached on the communicator. Thus, in this moment, the process 0 detects if there is more than one process per node. In that case, the process 0 chooses the process with smallest rank to become aggregator from all the processes located on the same node. This creates an ordered array of ranks (relative to the communicator used to open the file) that will be aggregators. This array is distributed to all processes using *ADIOI_cb_bcast_rank_map*. Aggregators are referenced by their rank in the communicator used to open the file. These ranks

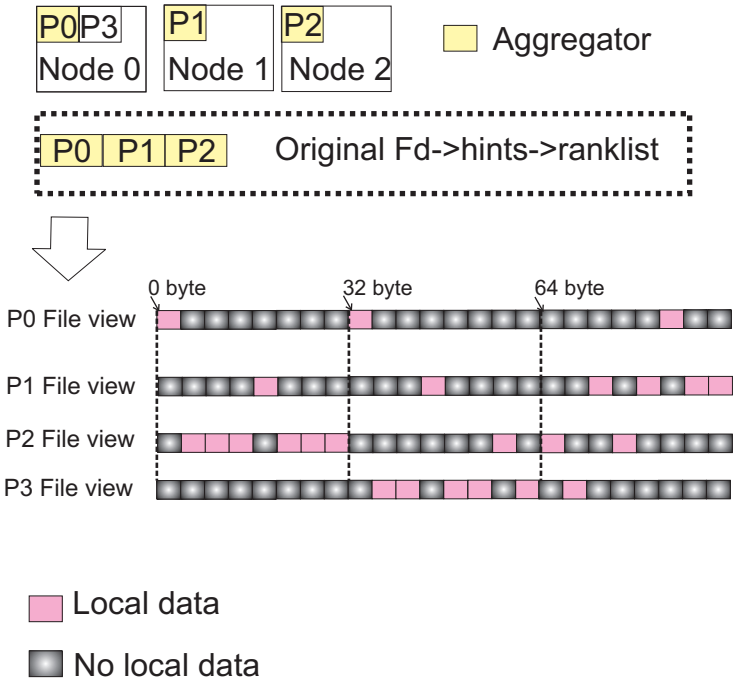


Figure 3.1: Default assignment of aggregators over processes.

are stored in an internal array called *fd ->hints ->ranklist[]*, that all processes have defined.

As follows, we illustrate the *Two-Phase I/O* technique through an example of a vector of 24 integers that is written in parallel by 4 processes (see Figure 3.1) to a file. The size of one element in this example is 4 bytes. Each process has previously declared a view on the file, i.e. non-contiguous regions are "seen" as if they were contiguous.

Before performing these two phases, process 0 has calculated the assignment of aggregators. Figure 3.1 shows that the process 0 has been selected as aggregator 0, the process 1 as aggregator 1, and finally process 2 as aggregator 2. The process 3 it is not selected because it is located in the same node as process 0.

Once the aggregators are assigned, each process analyzes which parts of the file are stored locally by creating two lists of offsets and lengths (*stage st1*). According to the example, process 0 is assigned three intervals: (*offset*=0, *length*=4), (*offset*=32, *length*=4), (*offset*=84, *length*=4). The list of offsets (offset_list) for this process is: {0, 32, 84} and the list of lengths (len_list) is: {4, 4, 4}.

In addition, each process calculates the first and last byte of the accessed interval. In our example, the first byte that process 0 has stored is 0 and the last one is 87. Next, all processes exchange these values and compute the maximum and minimum of file access range, in this case 0 and 95, respectively (*stage st2*). The interval is then divided into equal-sized FDs (*stage st3*). If all four processes are aggregators, it will be divided in 4 chunks of 24 bytes, one for each aggregator. But, in our case, only 3 processes are aggregators, thus the interval is divided in 3 chunks

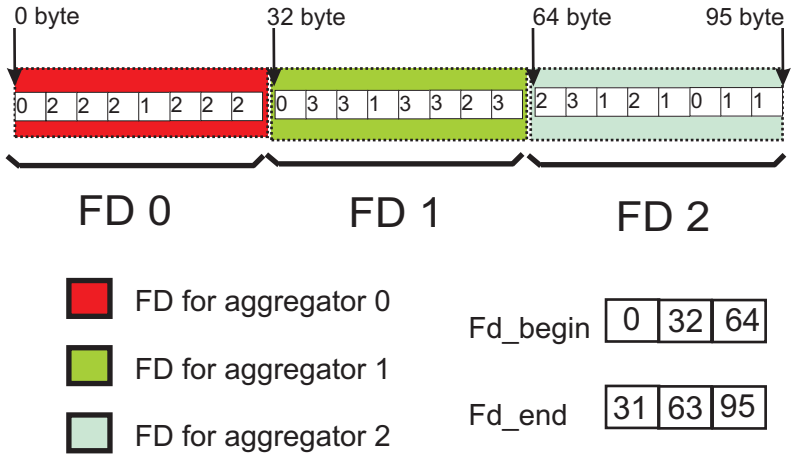


Figure 3.2: Division of file data into *file domains*.

of 32 bytes.

Each chunk is assigned to each aggregator according to the array *fd ->hints ->ranklist[]*. That is, block 0 is assigned to the process stored in fd ->hints ->ranklist[0] (in this case it is process 0), block 1 to fd ->hints ->ranklist[1] (process 1), etc. Each chunk (FD) is written to file by the assigned aggregator. For performing this operation, each aggregator needs all the data of its FD. If these data are stored in other processes, they are exchanged during the redistributed data exchange phase.

Once the size of each FD is obtained, two lists (called Fd_begin and Fd_end) with as many positions as number of aggregators are created. The Fd_begin list indicates the beginning of the FD of each aggregator. The Fd_end list indicates the end of the FD of each aggregator.

Figure 3.2 shows how the vector is divided into different FDs. Each FD has different colours assigned. Also, it can be observed that the assignment of FD is independent of the local data of each process. This scheme is inefficient in many situations. For example, the FD for aggregator 0, which has been assigned to process 0, begins at byte 0 and ends at byte 31. Most of these data are stored in process 2, so this implies unnecessary communications between process 0 and 2, because process 2 has to send all data to process 0, instead of writing them to disk. It would be better if the aggregator 0 was assigned to process 2 instead of process 0.

Once each aggregator knows all the referring data, it analyzes which data from its FD are not locally stored and what communication has to be established in order to gather them (*stage st4*). This stage is reflected in Figure 3.3. The arrows represent communication operations between the processes.

For example, in Figure 3.3, aggregator 0 (in our example process 0) needs seven elements that are stored in the processes 1 and 2, and has two elements stored that aggregators 1 and 2 need.

In addition, if transferred data are not contiguously stored, the process has to send as many messages as contiguous regions. This behaviour can be seen in

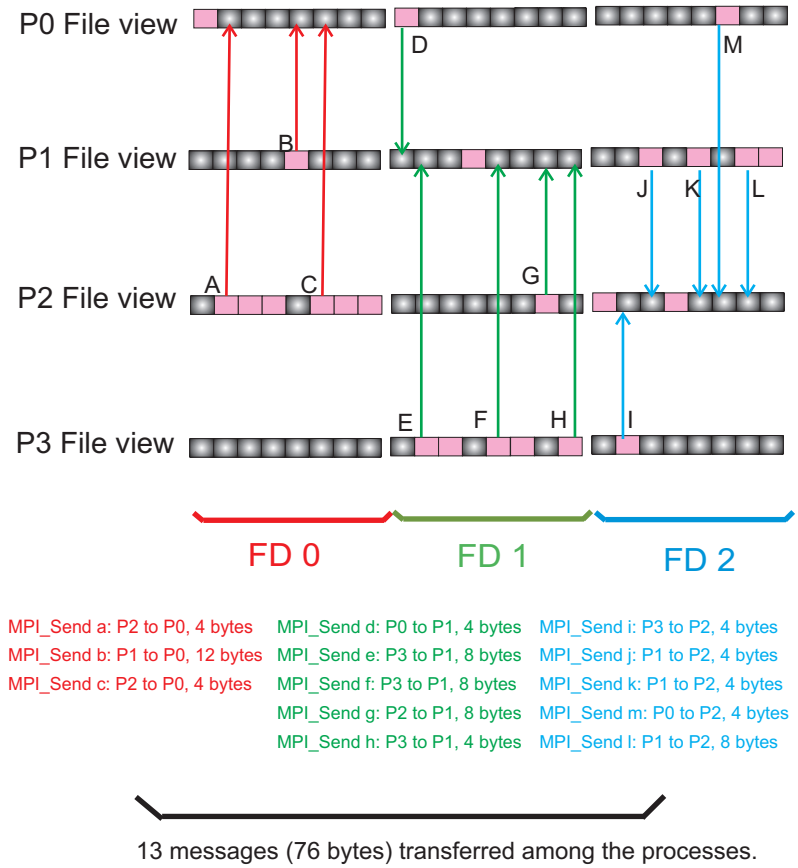


Figure 3.3: Data transfers among processes.

Figure 3.3, when the process 2 has to send two messages to aggregator 0 to transfer two intervals of data.

In the following step of *Two-Phase I/O*, the processes exchange the previous list of request of data (*stage st5*), before that the data are sent to appropriate aggregator(*stage st6*). Once all the aggregators have the data of their FD, the I/O phase begins and they write a chunk of consecutive entries to the file, as shown in Figure 3.4(*stage st7*). Each aggregator transfers only one contiguous region to the file (its FD).

We propose replacing this assignment by a new dynamic I/O aggregator pattern that decides how to distribute I/O aggregators among all process groups according to the local data that each process stores. Section 3.3 describes this technique.

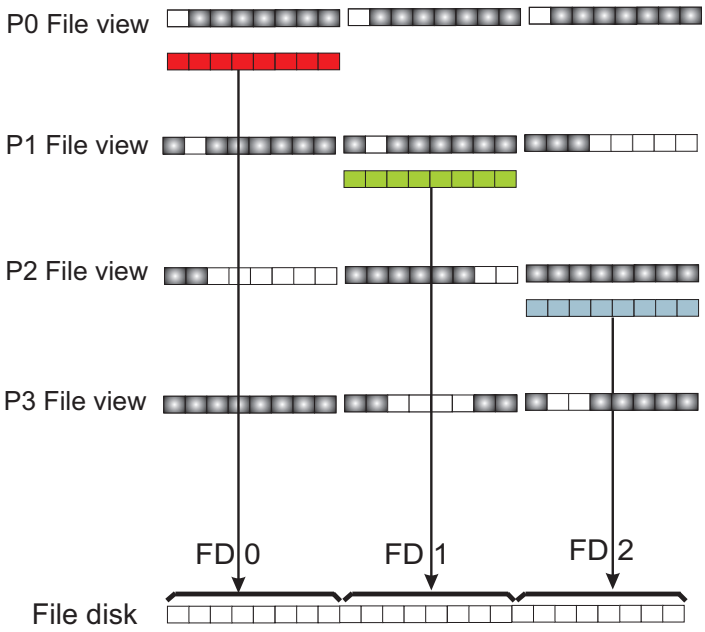


Figure 3.4: Write of data in disk.

3.3 Dynamic I/O Aggregator Pattern proposed for Two-Phase I/O

In this section, we explain our proposal to reduce the number of communications produced by *Two-Phase I/O*. As we explained in Section 3.1, we propose to replace the fixed aggregator pattern of *Two-Phase I/O* by a dynamic one based on specific data distribution over the processes. The new dynamic I/O aggregator pattern proposed, has as its main goal the reduction of the number and volume of communications involved in the redistribution phase of *Two-Phase I/O*. Therefore, with the new pattern, each aggregator does not need to perform as many requests to collect all data belonging to its file domain, because most of data are locally stored. With this reduction of communications, we aim to reduce the overall execution time and increase the performance and scalability of applications.

The new aggregator pattern proposed is dynamic, because it is calculated at runtime. In addition, each application has its own pattern. This means, that for some applications, the new pattern can assign to process 0 aggregator 0, and for others it could be any other process, and the same for the rest of aggregators. Moreover, the number of processes also affects to pattern. For example, if we increase or reduce the number of processes with which we execute an application, the distribution of data over the processes changes. Therefore, the new aggregator pattern also changes dynamically if we execute the applications with different configurations of processes.

We have developed an optimization of *Two-Phase I/O* called *LA_ TwoPhase I/O*, where we have implemented the new dynamic I/O aggregator pattern proposed. The internal strucuture of *LA_ TwoPhase I/O* is shown in Section 3.5.

3.4 Linear Assignment Problem

As explained in Section 3.2, *Two-Phase I/O* makes a fixed assignment of the aggregators over the processes. We propose replacing this fixed assignment of aggregators by a dynamic one. Therefore, to solve the problem of finding the best aggregator pattern we propose appling the existing solutions of *Linear Assignment Problem*.

The *Linear Assignment Problem* (LAP) is a well-studied problem in linear programming and combinatorial optimization. LAP computes the optimal assignment of n items to n elements given an $n \times n$ cost matrix. In other words, LAP selects n elements of the matrix, so that there is exactly one element in each row and one in each column, and the sum of the corresponding costs is maximum.

In our case, the LAP tries to assign the aggregators to processes, by maximizing the cost, given that we want to assign the aggregator to the process, for which it has more number of contiguous data blocks. For example, for the previous example, process 3 has stored three contiguous data blocks from FD 1. This means, that process 3 is the best candidate to be aggregator 1.

A large number of algorithms, both sequential and parallel, have been developed for LAP. We have selected the following ones for our work, which are considered to be the most representative:

- *Hungarian algorithm* [Bla86]: This is the first polynomial-time primal-dual algorithm that solves the assignment problem. The first version was invented and published by Harold Kuhn in 1955 and has a $O(n^4)$ complexity. It was revised by James Munkres in 1957, and has been known since as the Hungarian algorithm, the Munkres assignment algorithm, or the Kuhn-Munkres algorithm.

- *Jonker and Volgenant algorithm*[RA87]: They developed the shortest augmenting path algorithm for the linear assignment problem. It contains new initialization routines and a special implementation of Dijkstra's shortest path method. For both dense and sparse problems, computational experiments show that this algorithm is uniformly faster than the best algorithms from the literature. It has a $O(n^3)$ complexity.

- *APC and APS Algorithms*[CP88]: These codes implement the Lawler $O(n^3)$ version of the Hungarian algorithm by Carpenato, Martello and Toth. APC works on a complete cost matrix, while APS works on a sparse one.

3.4.1 Performance of the Linear Assignment Problem

As we explained before, there are several algorithms developed for LAP. In order to know what the best LAP algorithm is, we have evaluated all of them using a MPI parallel application, called *BIPS3D*, which uses collective write. This application it is described in Section 6.2.

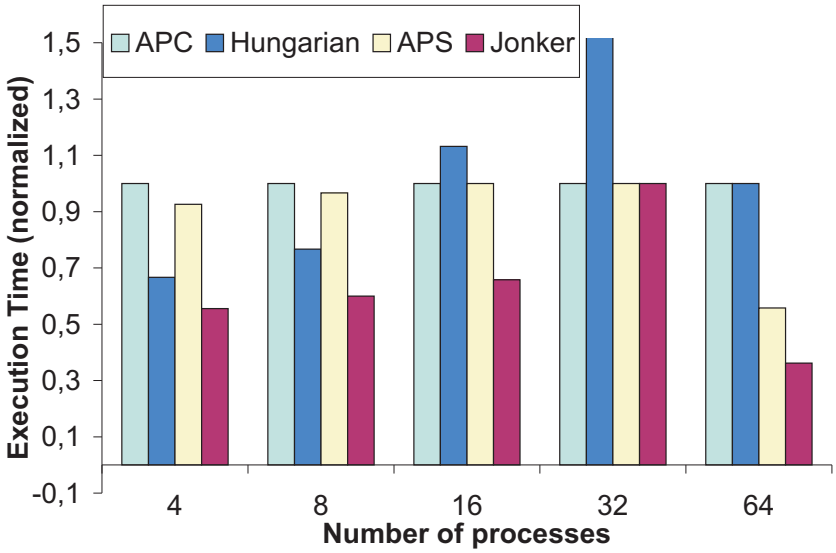


Figure 3.5: Time for computing the optimal aggregator pattern by using different LAP algorithms.

We have noticed that in all cases all LAP algorithms produce the same assignment (of aggregators). The only difference between them is the time to compute the optimal pattern. Figure 3.5 shows the normalized execution time (taking the APC algorithm as the reference technique) for calculating the new aggregator pattern. The x axis represents the number of processes (aggregators) involved in the collective I/O operation. That is, the problem size subjected to be optimized. Note that the fastest algorithm is the Jonker and Volgenant one, and for this reason we have chosen it for our proposal.

3.5 Internal Structure of LA_TwoPhase I/O

As we explained in Section 3.3, we have developed an optimization of *Two-Phase I/O* called *LA_ TwoPhase I/O*, where we have implemented the new dynamic I/O aggregator pattern proposed in this Ph.D. thesis.

In order to compute the best pattern, it is essential to know the distribution of data over the processes. The first version of *LA_ TwoPhase I/O* [FSP+08] was dependent of the application because the appliaction was responsible for calculating the distribution of data and for communicating it to *Two-Phase I/O*. Currently, the *LA_ TwoPhase I/O* version presented in this chapter is completely independent of the application because it is able to obtain this information at run-time.

The aggregator pattern of *LA_ TwoPhase I/O* is calculated in a new stage, once the file domains are obtained. This is because, in the optimization of *Two-Phase I/O* that we present in this chapter, each aggregator is assigned to the process which has the highest number of contiguous data blocks of the file domain associated with the aggregator. So, firstly *LA_ TwoPhase I/O* needs to know the characteristics of

each file domain (*stage st3*), and then, it calculates the best aggregator pattern by applying LAP problem. The rest of stages of *LA_ TwoPhase I/O*, are the same as *Two-Phase I/O*. As follows, we summarize the *LA_ TwoPhase I/O* stages:

- Previous calculations:
  - *Offsets and lengths calculation (st1).*
  - *Offsets and lengths communication (st2).*
  - *File domain calculation (st3).* associated to its FD.
  - *Dynamic Aggregator Pattern (st4)*: Each process calculates the number of data that has been locally stored for each FD. After that, each aggregator is assignated to processes by applying *Linear Assignment Problem.*
- Redistributed Phase:
  - *Access request calculation (st5).*
  - *Metadata transfer (st6).*
  - *Buffer writing (st7).*
- I/O Phase:
  - *File writing (st8).*

Figure 3.6 shows the aggregator pattern pseudocode corresponding to the *stage st4* of *LA_ TwoPhase I/O*. The MPI function calls within the pseudocode have been abbreviated for simplicity. A short description of the meaning of the fields within the pseudocode MPI function calls is therefore necessary:

- MPI_Send(item_send,number of items, data type of items, rank_destination)
- MPI_Recv(item_recv,number of items, data type of items, rank_source)

The pseudocode of Dynamic I/O aggregator pattern is divided into four phases: the calculation of which portion of data belongs to each file domain, recollection of distribution of data, calculation of best aggregator pattern, and distribution of new aggregator pattern. In the first phase, each process calculates in parallel, for all its offsets (*Label L1*), in what file domain they belong. In order to get this information, each process calls to *ADIOI_ Calc_ aggregator* function (*Label L2*). *ADIOI_ Calc_ aggregator* is an ADIO function that allows calculating in what file domains the data blocks that a process has stored are located. As a result, a vector per process is built (called *Array_ data*) (*Label L3*). This vector stores the number of contiguous data blocks that each process stores for each file domain.

The second phase (recollection of distribution of data) consists in gathering all *Array_ data*. Once the processes have built their *Array_ data*, each process sends this vector to process 0 (*Label 4* and *Label 5*). Then, process 0 builds the *assignment-matrix* gathering all arrays (*Label 6* and *Label 7*).

```
Begin parallel algorithm of dynamic I/O aggregator pattern:

input: num_fd  Number of File domains
       num_proccess  Number of processes
       contig_acces_count  Number of offset that each process has stored
       offset_list  List of offset that each process has stored
       len_list  List of lengths of offset that each process has stored
       rank  identifier of each process
       name_fd  identifier of each File domain

output: fd->hints->ranklist  List of processes that are selected as aggregators

CALCUATION WHICH PORTIONS OF DATA BELONGS TO EACH FILE DOMAIN
    for each rank ∈ num_process
L1    for each offset ∈ contig_access_count
L2      name_fd ← ADIOI_Calc_Aggregator(offset_list[offset], len_list[offset])
L3      array_data[name_fd] + +
      end for
    end for

RECOLECTION OF DISTRIBUTION DATA
L4   if(rank ≠ 0)
L5     MPI_Send(array_data, num_fd, integer, 0)
     else
L6     for each rank ∈ num_process
L7       MPI_Recv(assignment_matrix[rank], num_fd, integer, rank)
       end for
    end if

CALCULATION OF BEST AGGREGATOR PATTERN
L8  if(rank == 0)
L9    ranklist_new ← Lap_Jonker(assignment_matrix, num_process)
    end if

DISTRIBUTION OF NEW AGGREGATOR PATTERN
L10  if(rank == 0)
L11     if(franklist_new ≠ fd -> hints -> ranklist)
L12        fd -> hints -> ranklist = ranklist_new
L13        ADIOI_Cb_Bcast_Rank_Map(fd -> hints -> ranklist))
        end if
    end if


 End algorithm
```

Figure 3.6: Dynamic I/O aggregator pattern pseudocode.

The third phase (calculation of best aggregator pattern) obtains the new aggregator pattern. The criterion used for the new pattern is to maximize the aggregator locality by applying the LAP algorithm to the *assignment-matrix* (*Label 8* and *Label 9*). The LAP algorithm returns a list with the optimal assignment of aggregators over the processes (called *ranklist_ new*).

Finally, the last phase of pseudocode works as follows: Once the allocation has been obtained, process 0 checks if the new aggregator pattern is different from the original one (*Label 10* and *Label 11*). In that case, process 0 sends the new assignment of aggregators to other processes using *ADIOI_ cb_ bcast_ rank_ map* (*Label 12* and *Label 13*). In contrast, if both aggregator patterns are the same, it is not necessary to send the pattern.

As follows, we illustrate the *LA_ TwoPhase I/O* technique by using the same

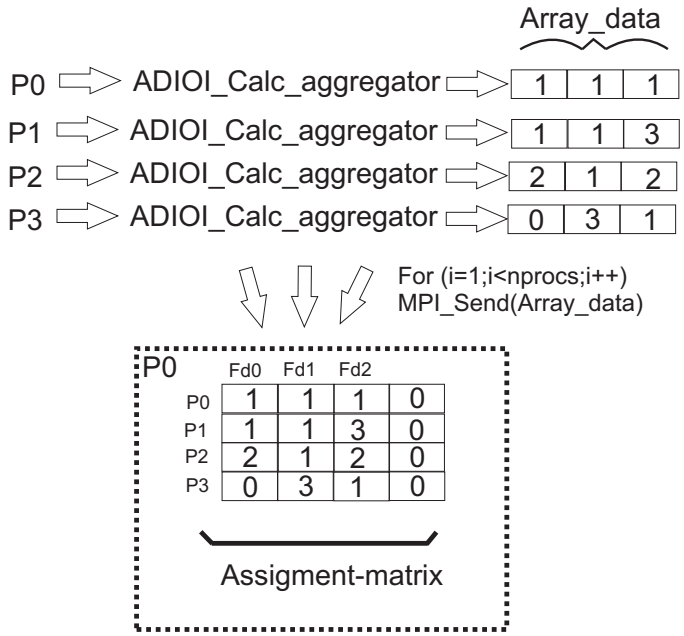


Figure 3.7: Calculating which portions of each process are located in which file domains.

example in Section 3.2 with the same data and distribution as in Figure 3.1. In our example, if we use the original aggregator pattern, aggregator 0 (process 0) needs to receive 7 data blocks of its FD, aggregator 1 (process 1) needs to receive 7 data blocks, and aggregator 2 (process 2) needs to receive 6 data blocks. That means there are 20 of the 24 data blocks which must be transferred among the processes to appropriate aggregators in redistribution phase.

Before the redistribution phase (*stages st5, st6* and *st7*), *LA_ TwoPhase I/O* searches the best aggregator pattern applying LAP algorithm (*stage st4)*. Process 0 is responsible for building the *assignment-matrix* and assigning each file domain to each process. Each process only knows how many contiguous data blocks of each file domain has been stored. Thus, each process has to calculate this information and send it to process 0 in order to build the *assignment-matrix*, as shown in Figure 3.7.

Finally, the LAP algorithm is applied to the *assignment-matrix* and it returns a list with the optimal assignment of aggregators over the processes, as shown in Figure 3.8. The new aggregator pattern for this example is:

- Aggregator 0 -> Process 2
- Aggregator 1 -> Process 3
- Aggregator 2 -> Process 1

The new dynamic I/O aggregator pattern reduces the number of communication operations because each aggregator increases the amount of locally assigned data. For this example, with the new aggregator pattern, the aggregator 0 (process 2)

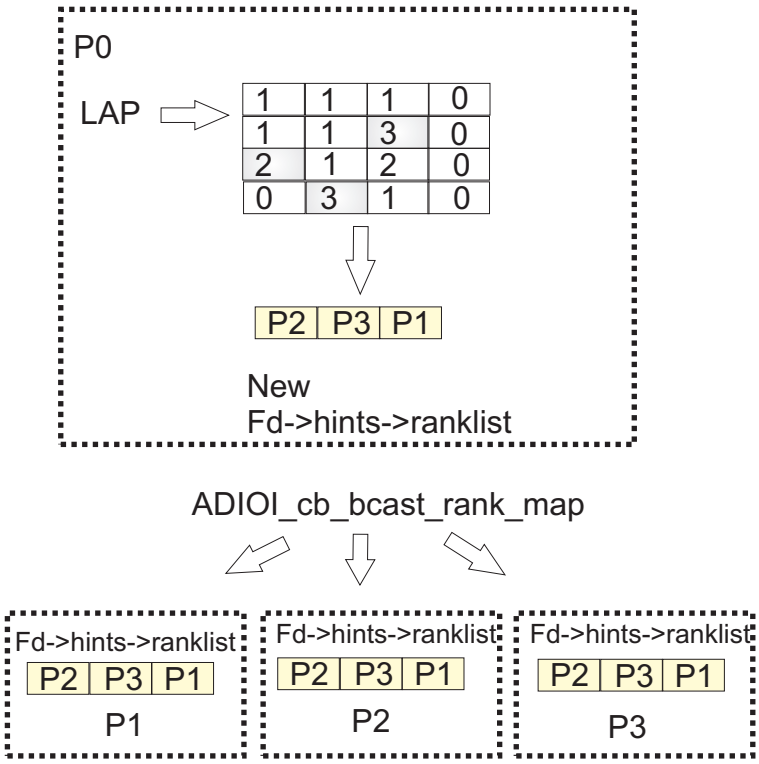


Figure 3.8: Example of building the *assignment-matrix* to obtain the new I/O aggregator pattern.

only needs to receive 2 contiguous data blocks, aggregator 1 (process 3) needs 3 contiguous data blocks, and aggregator 2 (process 1) needs to receive 4 contiguous data blocks of its FD, as shown in Figure 3.9. In this example, *LA_TwoPhase I/O* reduces the number communications from 13 to 9 comparing with *Two-Phase I/O*. In addition, the volume of communications is also reduced. With the original aggregator pattern, 76 bytes are transferred, and with the new one, only 36 bytes are transferred among the processes.

Once all the aggregators have their data stored, they write them into a file in chunks of consecutive entries, as shown in Figure 3.10. Note that the calculation of new I/O aggregator pattern is performed only once (*stage st4*). Instead, the exchange of data among processes and the write of data in disk can be performed several times (*stages st7 and st8*), depending on the FD size of each aggregator. By default, each aggregator writes only 4MB of data. Therefore, if the FD size is greater than 4MB, the aggregator has to perform several writes into the file, which may include more redistribution stages to request the data that each aggregator requires. So, with the proper I/O aggregator pattern, the number of communications is reduced throughout all stages of the redistribution phase.

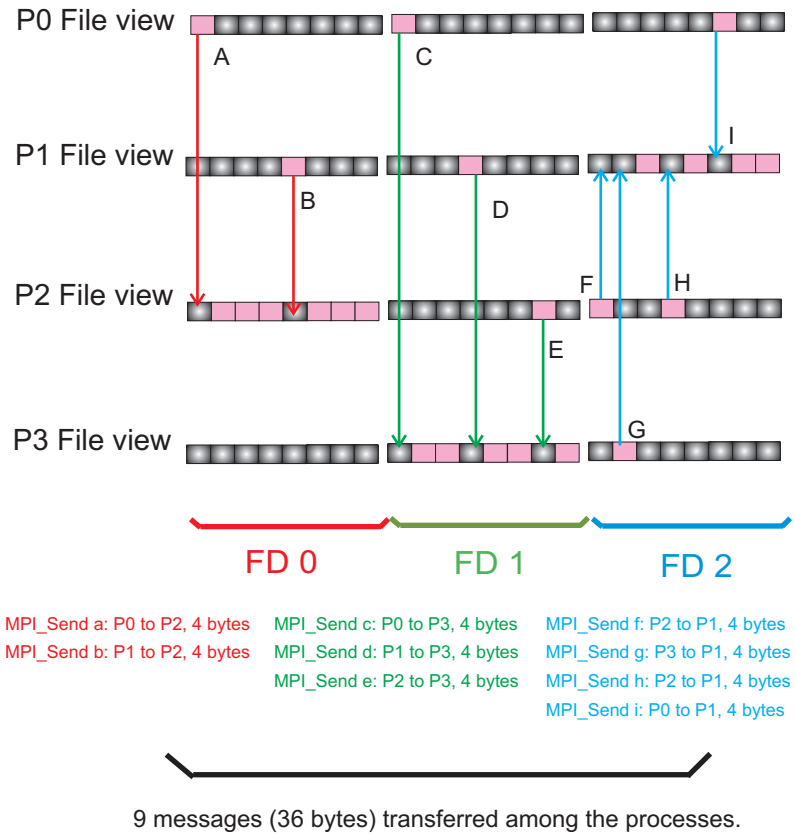


Figure 3.9: Data transferred with the new I/O aggregator pattern.

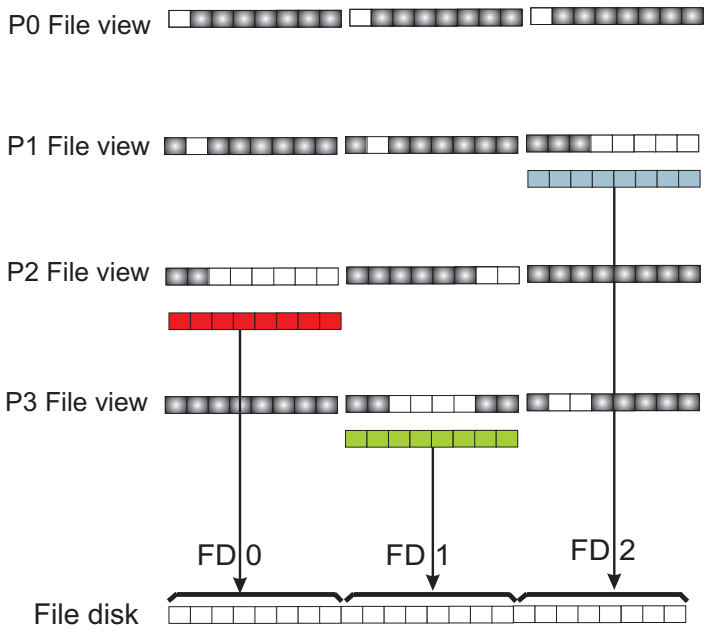


Figure 3.10: Write of data in disk with the new I/O aggregator pattern.

.

3.6 Summary

In this chapter we have proposed the optimization of the *Two-Phase I/O* collective technique from ROMIO. The goal of our optimization is to reduce the number of communications performed among the processes to increase the scalability and performance of parallel applications. In the default implementation of *Two-Phase I/O*, the aggregator pattern is fixed. This aggregator pattern might create a I/O bottleneck, as consequence of the multiple requests performed by aggregators to collect all data assigned to their FD.

We propose a new aggregator pattern based on local data that each process stores. To find the optimal aggregator pattern for each application in run-time, we have used the *Linear Assignment Problem* (LAP) . The result, is a new technique called *LA_ TwoPhase I/O*, where we have implemented the new dynamic I/O aggregator pattern, as a new step of *Two-Phase I/O*. In the *LA_ TwoPhase I/O* technique, the number of communications involved in the redistribution phase is reduced, as shown in Section 3.5.

Furthemore, our proposal can be used for any kind of application that uses *Two-Phase I/O*, in a transparent way for the user and without any modification of the source code.

Chapter 4

Enhancing MPI Applications by Using Adaptive Compression

4.1 Introduction

Parallel computation on cluster architectures has become the most common solution for developing high-performance scientific applications. Many scientific applications require a large number of computing nodes and operate on a huge volume of data that has to be transferred among the processes using messages.

When measuring the performance of parallel applications on clusters, the communication system is always one of the major limiting factors. Commodity clusters, where networks such as Fast Ethernet or Gigabit are installed, have high latency and low bandwidth, thus becoming a bottleneck affecting performance. Scalability is also an important issue in these systems when many processors are used, which may cause network saturation and still higher latencies. As communication-intensive parallel applications spend a significant amount of their total execution time exchanging data between processes, the former problems may lead to poor performance in many cases.

For theses reasons, in this chapter we have focused on reducing the transferred data volume to dismiss the communication bottleneck. We propose reducing the cost of interchanged messages, diminishing the data volume by using lossless compression among processes.

Lossless message compression is a technique that has been commonly used for reducing communications overhead in clusters [DLY+98]. Therefore, we propose a new strategy called *Runtime Adaptive Compression*, that uses message compression in the context of large-scale parallel message-passing systems to reduce the communication time of individual messages and to improve the scalability and performance

of applications executed in clusters.

Scientific applications usually transfer different types of messages (diferent data type, length etc...) among the processes. In this way, the *Runtime Adaptive Compression* strategy, integrates different compression algorithms that can be selected at run-time taking into account the specific communication characteristics (type of data, size of the message, etc.), the platform specifications (network latencies, computing power, etc.) and the compression technique performance.

Note that the new compression strategy is transparent for users. Thus, it selects dynamically the best compression algorithm per message, with the least overhead possible, and without modifications over the applications.

The compression in *Runtime Adaptive Compression* is applied to all communications of MPI, including the ones performed in I/O subsystem. Furthermore, *the Runtime Adaptive Compression* strategy is able to turn itself on and off, because it is not always efficient to compress all the messages from an application. Thus, in some cases the best choice is to not compress.

4.2 Using Compression in MPI

As we explained before, in this chapter we focus on reducing the volume of the transferred messages among the processes. Therefore, we initially test the feasibility of using run-time compression in MPI messages, developing a strategy called *Runtime Compression* that compresses all the messages of an application with the compression algorithm indicated by the user.

Runtime Compression strategy addresses all types of communications, and includes different compression techniques (LZO, RLE, HUFFMAN, RICE and FPC) that can be used transparently to users (by means of MPI hints). However, in *Runtime Compression* strategy, compression not can be disabled, and the compression algorithm must always be the same to compress all the transferred messages. We propose evaluating this strategy in the *MPICH* [GL97] implementation.

The architecture of *MPICH* consists of 3 layers: Application Programmer Interface (API), Abstract Device Interface (ADI) and Channel Interface (CI). The API is the interface between the programmer and ADI. It uses an ADI to send and receive information. The ADI controls the data flow between API and hardware. It specifies whether the message is sent or received, handles the pending message queues, and contains the message passing protocols. The Channel layer defines three protocols to send messages based on the message size: short, eager (long) and rendez-vous (very long). MPICH chooses the protocol based on the message length (in bytes).

Note that the ADI layer is a portable layer, while the Channel layer is not. Therefore, we propose modifing the ADI layer in order to include the *Runtime Compression* strategy, as shown in Figure 4.1, independently of the channel and protocol used. *Collective* communication routines are implemented using *point-to-point* routines. Therefore, applying *Runtime Compression* strategy on point-to-point

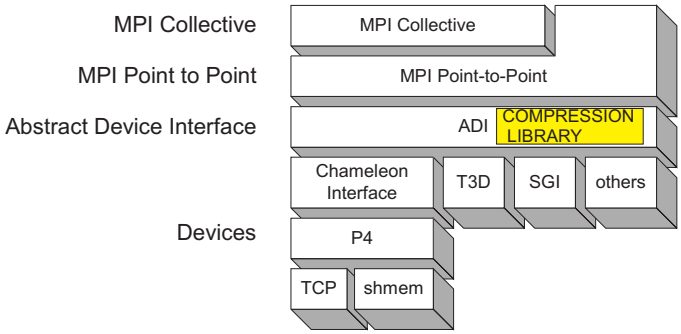


Figure 4.1: Layers of *MPICH*.

routines, not only compresses these communications, but also the collective ones. All compression and decompression algorithms used by *Runtime Compression* strategy are collected in a single library called *Compression-Library*. Thus, as ocurs with *Runtime Compression* strategy, we propose locating this library in the ADI layer.

We have also taken into account all types of communications that MPI supports: *blocking* and *non-blocking*. The blocking message does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. Instead, non-blocking communications allow the overlap of computations and communications, because they enable the user to use the CPU even during ongoing message transmissions at the network level. In this way, the *Runtime Compression* strategy compresses a message before the process sends the message, as shown in Figure 4.2, independent if it is a blocking or non-blocking communication, and it decompresses a message after the receiver posts an `MPI_receive` operation, and also copies the data into its buffer. We want to highlight that the compress and decompress operations are performed in two different places, depending if the message is blocking or non-blocking.

- Compress Operation:
  - Non-blocking compression: Before that MPICH executes a non-blocking send, such as `MPI_Isend`, the message compression is performed.
  - Blocking compression: As a non-blocking send, before MPICH executes a blocking send, such as `MPI_send`, the message compression is performed.
- Decompress Operation:
  - Non-blocking decompression: When MPICH executes a non-blocking receive, such as `MPI_Irecv`, the message decompression (expected or unexpected) is performed after data communication is complete. These means, when `MPI_Wait` is called.

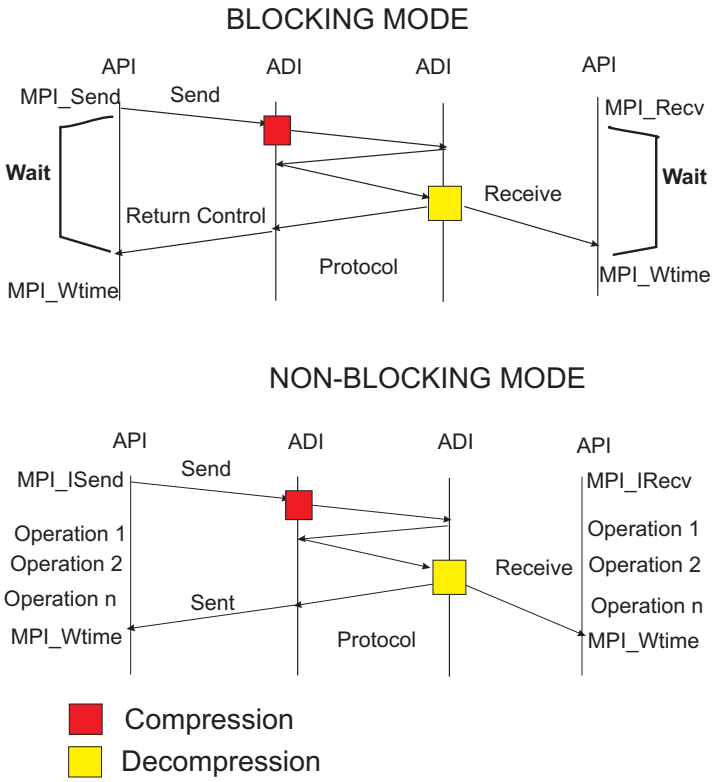


Figure 4.2: *Runtime Compression* strategy for Blocking and Non-Blocking communications (rendez-vous protocol).

– Blocking decompression: In the other case, in blocking receive, the message decompression is performed when the receiver finished receiving the complete message (after *MPI_Recv_Complete* is finished).

The basic MPI communication mechanisms (blocking and non-blocking communications) can be used to send or receive a sequence of elements that are contiguous in memory (*contiguous messages*). However, it is often desirable to send data that is not contiguous in memory (*non-contiguous messages*). Both types of messages are also considered in *Runtime Compression* strategy. In the case of contiguous datatypes, the whole set of transferred data is subjected to compression. When non-contiguous datatypes are employed, the compression is applied after the data packing. Then, data are subsequently uncompressed before being unpacked. This approach allows all the possible datatypes to be handled transparently.

The target of *Runtime Compression* strategy is to compress in runtime all types of communications explained before: point-to-point, collective, blocking, non-blocking, contiguous and non-contiguous messages. In this way, the *Runtime Compression* strategy distinguishes between blocking or non-blocking communications, and also between contiguous or non-contiguous messages, and applies in each case the *Runtime Compression* strategy.

To achieve a benefit from transferring compressed instead of decompressed data, the additional computational time of the compression algorithm has to be lower

```
 begin Send_Message_Blocking pseudocode {

input:  buff buffer where is located the data to send
        *dtype_ptr  pointer of data type of data
         dest_rank  identification of process receiver
         src_rank  identification of process sender
         count  Number of element to send

 CHECK IF DATA ARE CONTIGUOS OR NOT
L1   conitg_size ← Get_datatype_size(dtype_ptr)

 COMPRESS AND SEND CONTIGUOUS DATA IF BUFFER > 2048 BYTES
L2  if(contig_size > 0)
L3     len_buffer = contig_size * count
L4     if(len > 2048)
L5        algorithm_compress ← Read_Hint_User()
L6        len_compress, buff_compress ← Compression_Message(buff, len, algorithm_compress)
L7        Send_Message_contiguous(buff_compress, len_compress, src_rank, dest_rank)

 SEND CONTIGUOUS DATA IF BUFFER <= 2048 BYTES
         else
L8           flag_compress = no_compress
L9           buff ← Add_Head(flag_compress, 0, buff)
L10          Send_Message_contiguous(buff, len, src_rank, dest_rank)
         end if

 COMPRESS AND SEND NON-CONTIGUOUS DATA IF BUFFER > 2048 BYTES
      else
L11       len_packet, buff_packet ← Pack_Message(buff, len)
L12       if(len_packet > 2048)
L13          algorithm_compress ← Read_Hint_User()
L14          len_compress, buff_compress = Compression_Message(buff_packet,
             len_packet, algorithm_compress)
L15          Send_Message_contiguous(buff_compress, len_compress, src_rank, dest_rank)

 SEND NON-CONTIGUOUS DATA IF BUFFER <= 2048 BYTES
              else
L16              flag_compress = no_compress
L17              buff_packet ← Add_Head(flag_compress, 0, buff_packet)
L18              Send_Message_contiguous(buff_packet, len_packet, src_rank, dest_rank)
              end if
     end if
 End algorithm}
```

Figure 4.3: Send_message_blocking pseudocode in *Runtime Compression* strategy.

than the time saved during the communication. We have performed several studies to decide the minimum message size threshold to apply compression efficiently. The results show that if a message is smaller than 2 KB more time is spent performing the compression/decompression than sending the decompressed data. For this reason, when the message is smaller than 2 KB, we do not apply compression in *Runtime Compression* strategy. As follows we show in Figures 4.3 and 4.4 the pseudocodes of Send_Message_Blocking and Receive_Message_Blocking of *Runtime Compression* strategy, which corresponds to how the strategy works with blocking operations by using contiguous and non-contiguous messages.

```
begin Receive_Message_Blocking pseudocode {

input:  buff buffer where is located the data to receive
        *dtype_ptr  pointer of data type of data
         dest_rank  identification of process receiver
         src_rank  identification of process sender
         count  Number of element to receive

 CHECK IF DATA ARE RECEVEIVE COMPLETE
L1     Recive_Data(buf, count, src_rank, dtype_ptr, request)
L2     check_request ← MPID_RecvComplete(request)

 CHECK IF DATA ARE COMPRESSED
L3     if(check_request == 1)
L4        flag_head = Study_Head_of_Buffer(request.buf)
L5         if(flag_head == yes_compression)


 DECOMPRESS THE BUFFER
L6            buf ← Decompression_Message(request.buf)
            else
L7               Copy(request.buf → buf)
            end if
        end if
 End algorithm}
```

Figure 4.4: Receive_Message_Blocking pseudocode in *Runtime Compression* strategy.

Send_Message_Blocking algorithm in *Runtime Compression* strategy has the following steps: Firstly, the algorithm checks if the message to send is contiguous or not (*Label L1*). If the message is contiguous (*Label L2*), the next step is checking the size of message (*Label L3*). If the message size is larger than 2KB (*Label L4*), the message is compressed with the algorithm indicated by the user in MPI_hint (*Labels L4* and *L5*). Secondly, once the message is compressed, the *Runtime Compression* strategy sends the *message in blocking mode* (*Label L7*). While if the message is smaller than 2KB, the message is sent in blocking mode without compression (*Label L10)*, but before sending it, a header is added (*Labels L8* and *L9*) to the message in order to indicate to receive process that it must not to decompress the message. Finally, if the message is non-contiguous type, the message is packed (*Label L11*) in a new buffer, and then, similar steps are done with the packed buffer (since *Labels L11* to *L18*).

Note that if the message to send is a non-blocking communication, the algorithm is similar to Send_Message_Blocking algorithm, but in this case the data are sent using Isend_Message_Contiguos, instead of Send_message_contiguous (*Labels L7,L10,L15* and *L18*). The rest of the steps are the exactly the same as explained before in pseudocode of Figure 4.3.

The algorithm to receive a message in blocking mode for *Runtime Compression* strategy has the following steps: The first step of this algorithm is receiving the complete message (*Labels L1,L2* and *L3*). When the receive process has stored the

complete message in its buffer, it studies the head of the message (*Label L4*) to know if it has to decompress the message or not, because the message can be sent without compression (if the message is smaller than 2KB). In case where the the message is sent compressed, the receive process has to decompress it (*Label L5* and *Label L6*). For receiving messages in non-blocking mode, the algorithm in *Runtime Compression* strategy is similar to the blocking mode algorithm. The difference regarding the former proposal is that decompression in *Runtime Compression* strategy is applied after the *receive call* in case of blocking communication, and after the *wait call* in the other case.

In Section 4.8.1, we explain in detail the algorithms employed to compress and decompress messages (which means, the Compression_message and Decompression_message).

To evaluate this *Runtime Compression* strategy, we have integrated it into the *MPICH* [GL97] implementation, and the new implementation of MPICH with the *Runtime Compression* strategy proposed in this section called *CoMPI* technique. The evaluation results (see Section 6.5.1) show that, in most cases, the use of compression in all communication primitives is feasible. In addition, the performance gain is greater when more processes are employed. Although, the results also show that is not always efficient to compress all messages from an application, and it is not always efficient to use the same compression algorithm for each datatype.

For this reason, we propose a new compression strategy called *Runtime Adaptive Compression* strategy [FCS+10a], whose main feature is to select at run-time the best compression algorithm for each message and to decide whether it is worth compressing data or not.

4.3 Runtime Adaptive Compression Strategy

As we explained before, the main problem of *Runtime Compression* strategy is that, it can not deactivate when the compression is not worth it. Furthermore, all the messages are compressed with the same algorithm. Hence, we propose another compression strategy called *Runtime Adaptive Compression* (RAS), which learns in run-time from previous messages to choose the compression algorithm to be used and the minimum message size that allows a benefit to be obtained by sending the message compressed.

Figure 4.5 shows the new strategy architecture proposed. The core is composed of the *Runtime Adaptive CoMPI*. This module, included into MPI run-time, allows to adapt the compression strategy to be adapted the application behavior during program execution.

Our goal in the RAS strategy is to use the most appropriated compressor (including the null compressor) in order to maximize the speedup per message and to reduce the applications' execution time. This means that, the RAS strategy has to take two decisions before sending a message:

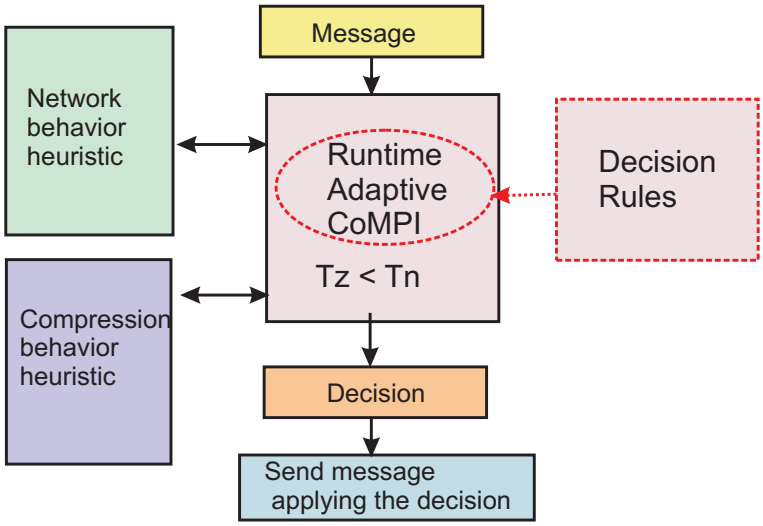


Figure 4.5: *Runtime Adaptive Compression* (RAS) strategy architecture.

1.- Selecting the most appropriated compression algorithm depending on the message features.

2.- Sending the messages compressed only when: The time of processing and transferring the message compressed is less than the time of transferring it uncompressed.

In order to include these functionalities, RAS strategy needs some mechanism that allows which compression algorithm to apply to be detected in run-time, and estimates the time to transfer the message with and without compression. For these reason, we have developed two modules in order to provide *Runtime Adaptive Compression* with a strategy with some network and compression information to take the former two decisions in the best way. The first module that we have built is *Network-behavior*. It estimates the latency and bandwidth in order to predict the time needed to send a message, generating a *Network-behavior heuristics* file for each installation. This file contains the network characteristics (latency and bandwidth) for each link between two communicating processes. The second module, called *Compression-behavior*, selects the best compression algorithm depending on the message datatype and its entropy level. Also, this model estimates the time needed to compress and to decompress a message with different compression algorithms (compression information). It also generates a *Compression-behavior heuristics* file for each installation. This file is used to decide which is the best algorithm in order to efficiently compress a message depending on the message features. By using those heuristics, *Runtime Adaptive Compression* strategy can choose the most appropriated compression algorithm, and it can also estimate the speedup per message, thus deciding to send it compressed or not, as shown in Figure 4.5.

After getting the former information, the RAS strategy must decide whether to send the message compressed or not. But, to do that, it has to predict the speedup generated by this operation. To predict the speedup, the RAS strategy needs to know

the message transmission time, and the message compression and decompression times. Message transmission time depends on the network latency, the amount of data to be transmitted, and the network bandwidth [MVCA97]. The total time (T_n) can be modelled by [KBS04b]:

$$T_n = T_{Latency} + \frac{message\_size}{bandwidth} \tag{4.1}$$

When data compression is used in data transmission, our main goal is to increase the transmission speed which also depends on the time needed to compress ($T_{compress}$) and decompress ($T_{decompress}$) the data. In this case, the total time (T_z) can be modelled by:

$$T_z = T_{compress} + T_{Latency} + \frac{compressed\_message\_size}{bandwidth} + T_{decompress} \tag{4.2}$$

With the former definitions (T_n and T_z) we can define the speedup as:

$$Speedup = \frac{T_n}{T_z} \tag{4.3}$$

As can be shown, $T_z < T_n$ if and only if:

$$T_{compress} + T_{decompress} < \frac{message\_size - compressed\_message\_size}{bandwidth} \tag{4.4}$$

That is, the time saved by sending a lower number of bytes is higher than the time needed to compress and decompress the message. Therefore, it is necessary to know five key values: $T_{compress}$, $T_{decompress}$, $compressed\_message\_size$, $bandwidth$, and $message\_size$.

Finally, after executing the RAS strategy, the sender process decides how to send the message. If compression is chosen, the original message is compressed using the algorithm selected. In any case, the MPI message buffer is modified to include a header in order to inform the receiver process whether it has to decompress the message or not, and which algorithm must be used. The implementation details are described in Section 4.8.

4.4 Network behavior modelling

As we explained in Section 4.3, we need to estimate the time to transmit a message using MPI primitives. The behavior of the MPI communication primitives depends on two main factors: the network technology used, and the communication software stack [MM05]. TCP/IP is the most popular stack in clusters. MPI implementation using the TCP/IP stack implements some different strategies for optimizing

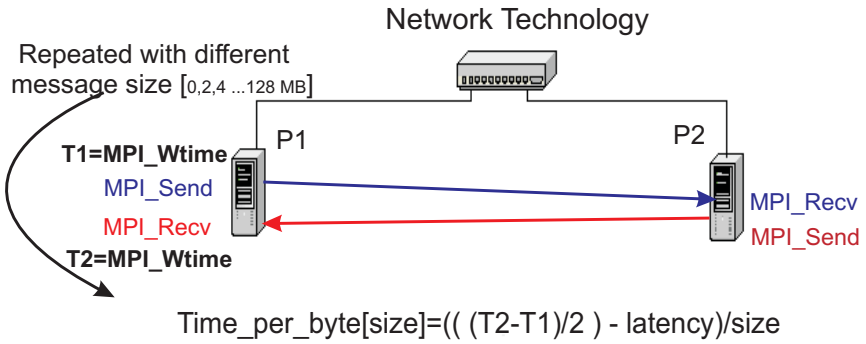


Figure 4.6: Network behavior test.

messages, such as early-eager or rendez-vous [RA08]. Some common network technologies used on clusters are: Ethernet, Gigabit, Myrinet and Infiniband.

Modern MPI implementations try to use novelty hardware (Infiniband for example) directly, by avoiding the TCP/IP stack overhead [LMP03]. Because several combinations of network technology and MPI software can be found, the transmission model has to be measured for each cluster taking into account the time needed by a MPI implementation for sending and receiving data of different sizes.

There are some interesting tests used for measuring the latency and bandwidth of any MPI implementation primitives. Examples of these are: MPPtest [GL99], MPBench [MLM98] and SkaMPI [RST02]. But, the requirement of being adaptive, to store the results in a specific table format, and the need to especially test point-to-point MPI primitives led us to write our own test because none of the previous ones satisfied the RAS requirements. Our test is called *Network-behavior*.

The *Network-behavior* is based on computing the bidirectional bandwidth between every pair of nodes, given a machine list in a cluster, as shown in Figure 4.6. In Chapter 6 it is reflected that many clusters are used in this thesis. As an example, we show the network model of one of them.

We have used the Hockney model [Hoc94] to model communication behavior. The Hockney model assumes that the time to send a message of size m bytes between two nodes is $\alpha+\beta*m$, where α is the latency for each message, and β is the transfer time per byte.

To apply the Hockney model we have calculated the values of β, m, and α. As

Size	2	4	8	16	32	64	128	256	512
Time_per_byte	6.463	4.269	1.347	0.719	0.446	0.328	0.280	0.230	0.242
Size	1024	2048	4096	8192	16384	32768	65536	131072	*T_Latency*
Time_per_byte	0.203	0.162	0.123	0.105	0.096	0.091	0.088	0.088	86.34

Table 4.1: Network-behavior transmission time (μsecs. per byte).

Figure 4.6 shows, the benchmark exchanges two messages for every different pair of nodes and with different message sizes, varying from 2, 4 bytes ... up to 128 MB (131072 bytes). The values are measured 100 times, average times are computed, and stored by *Network-behavior* in a heuristics table.

The results of the *Network-behavior* in our test platform, in microseconds per byte, are shown in Table 4.1. Therefore, α is called $T\_Latency$ in Network-behavior, and β*m is calculated using Table 4.1. Equation 4.5 allows the time needed to send a message in our cluster to be estimated.

$$Time\_to\_send\_message = (message\_size * Time\_per\_byte[message\_size]) + T\_Latency \quad (4.5)$$

Therefore, speedup can be calculated using equation 4.6:

$$Speedup = \frac{Time\_to\_send\_message}{Time\_to\_send\_message\_compressed + T\_compress + T\_decompress} \quad (4.6)$$

The measures produced by *Network-behavior* are stored in an internal Table called the *Heuristic Table*. When the test ends, the *Heuristic Table* is stored in a *Network behavior heuristics* file. This file is an input for the RAS strategy used in every MPI low level point to point communication in order to estimate the time to send a compressed and decompressed message.

4.5 Compression behavior modelling

Network characterization is only one part of the information needed by RAS strategy. Given a message, we want to select the most appropriated compressor, which is the one providing the lowest compression/decompression time ($T_{compress}/T_{decompress}$) and the highest compression ratio ($compressed\_size/original\_size$).

We have developed a synthetic test called *Compression-behavior*, whose structure is shown in Figure 4.7. This test selects the most appropriated compressor considering size, datatype, and entropy level (redundancy level) of a buffer.

The *redundancy level* of a given compression buffer (measured in percentage) is defined as the percentage of entries with the same numerical values. In this thesis we use this term instead of *entropy level*. In the next subsection, *redundancy level* is detailed with more precision.

When a process sends a message in the RAS strategy, it needs to select a compression algorithm and also it needs to calculate the speedup by using equation 4.6. Once the RAS strategy has selected the compression algorithm, and the $time\_to\_send\_message$ compressed and uncompressed are estimated using *Network-behavior*, the only value that is unknown in this equation is the decompressing time ($T_{decompress}$). The $T_{compress}$ can be measured in the sender process. Thus, it is only necessary to estimate the decompressing time. The *Compression-behavior* also ob-

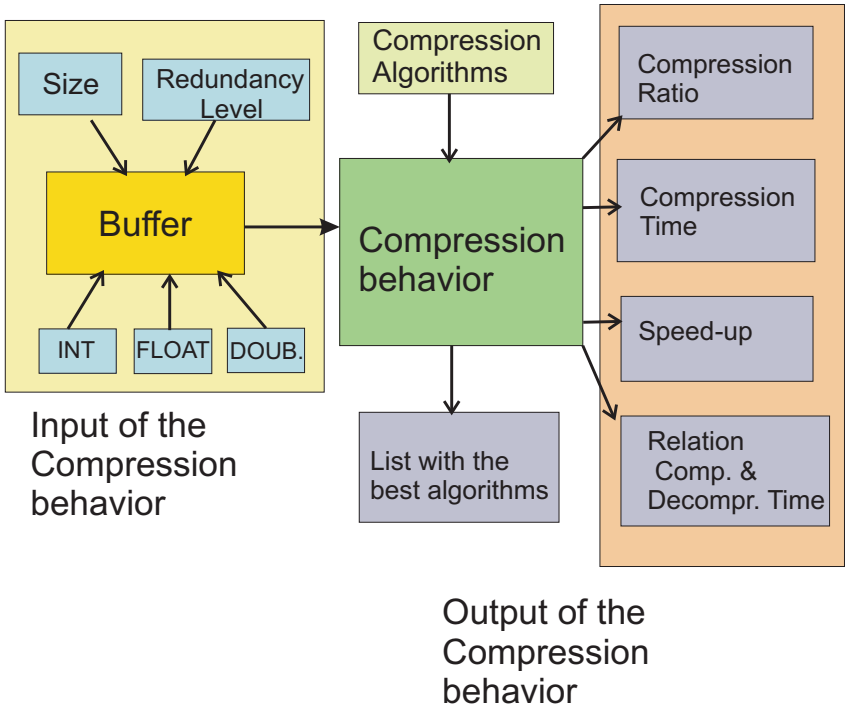


Figure 4.7: Compression behavior test.

tains the relation between the compression and decompression time for each algorithm.

In summary, the *Compression-behavior* studies four aspects:

- The time needed to compress and decompress different messages.
- The speedup achieved by using different messages and different compression algorithms.
- The compress ratio of different datatypes, redundancy levels, and compression algorithms.
- The relationship between the compression and decompression time for each algorithm.

The measures produced by *Compression-behavior* are stored in an internal table. When the test ends, the table is stored in a file called *Compression behavior heuristics*. This file is used, combined with the *Network Behavior heuristics* file, by the RAS strategy to choose the most suitable compression algorithm in every MPI low level point to point communications.

The rest of this section describes *Compression-behavior* features: selecting the best compressor, and defining the relationship between compression and decompression time.

4.5.1 Selecting the best compressor for each datatype

When data compression is used in data transmission, the goal is to increase the transmission speed which depends on the number of bytes sent, the time required for the compressor to generate the compressed message, and the time required for the decoder to recover the original ensemble. Therefore, the *Compression-behavior* studies the algorithm complexity and the amount of compression for each compressor with different types of data, buffer sizes, and redundancy levels.

As we explained before, *redundancy level* is defined as the percentage of entries with the same numerical values. In our case, the zero has been used as a data item. If the buffer redundancy level is 0%, it means that all values are different. But, if the redundancy level is 100%, it means that all values are equal to zero. These entries are randomly distributed among the buffer memory map. The rest of entries with different values are generated with random numbers.

For example, let us consider the following two sequences of ten integer values:

- Sequence 1: 0 3 2 18 14 9 4 3 0 12
- Sequence 2: 0 1 6 0 0 9 0 3 0 7

Sequences 1 and 2 have 20% and 50% redundancy levels, respectively. Note that the distribution of null values is random and it produces non-consecutive sequences of values. This solution is a worst-case scenario. In the literature there are solutions to generate random test patterns that use distributions for similar data grouping [Wun90],[Maj96].

We have developed synthetic datasets with three datatypes (integer, floating-point, and double precession). Each dataset contains buffers with different properties:

- Buffer size: The buffer sizes considered are 100 KB, 500 KB, 900 KB, and 1500 KB for integer and floating-point datasets. For double precision datasets, the buffer sizes are 200 KB, 1000 KB, 1800 KB and 3000 KB.
- Redundancy levels: 0%, 25%, 50%, 75% and 100%.

The *Compression-behavior* compresses the three datasets with the algorithms described in Table 2.1. As a result, the *Compression-behavior* builds a table with the compression ratio, and the compression/decompression time for each dataset (integer, floating-point and double precession). Table 4.2 shows the results obtained for integer datasets when different compression algorithms are applied to different buffer sizes, each one with a different redundancy levels. Each row of Table 4.2 has the compression information for each buffer size and different redundancy level in the dataset.

Once the tables are built, the *Compression-behavior* evaluates the speedup of each dataset by using the equation 4.6 in order to select the best compressor in each

case. The application estimates the time to send the original and the compressed messages with the information of Table 4.1 and equation 4.5.

Figure 4.8 illustrates the speedup of this evaluation for integer dataset. We can see that the LZO compression algorithm achieves the best results. Looking at the compression ratio (shown in Table 4.2), we can see that the Rice compression algorithm achieves the highest ratio when the redundancy level is between 0% and 75%. In contrast, when the redundancy level is 100%, the RLE and LZO compression algorithms achieve the highest compression ratios. However, Table 4.2 also shows that the Rice algorithm is slower than the LZO and RLE algorithms for all the scenarios considered. In terms of performance, we found that the LZO compression algorithm is the most appropriate to compress integer data.

We have done the same study for floating-point and double-precision datasets. Figure 4.9 shows the results in terms of the speedup for floating-point and Figure 4.10 for double-precision data. Figure 4.9 illustrates, for the redundancy level of 0%, that compression is not useful in order to improve the performance. For redundancy levels equal to or higher than 25%, LZO and RLE are the algorithms that obtain the best results.

We have evaluated an additional algorithm (FPC), for double-precision data, whose main characteristics are detailed in Section 2.4.5. The FPC is based on data prediction. In order to evaluate the effectiveness of the prediction algorithm, two different traces have been used: with and without pattern[1]. As Figure 4.10 shows, for traces without pattern, the LZO achieves again the best speedup. However, for

[1]We understand as a pattern a given data sequence that can be easily predicted. An example of this sequence is: 50001.0 50003.0 50005.0 50007.0 ...

Size (KB)	Redundancy Level %	Compression ratio				Time compr. + decompr. (ms)			
		RLE	HUFF	RICE	LZO	RLE	HUFF	RICE	LZO
100	0%	0	38.79	52.54	16.37	1.50	8.70	7.60	62.10
100	25%	22.73	50.28	63.06	36.22	1.50	6.90	7.10	1.70
100	50%	44.26	60.88	72.90	58.93	1.50	5.90	6.50	1.30
100	75%	81.77	78.87	89.3	84.73	1.50	3.20	6.0	0.70
100	100%	99.98	87.50	96.87	99.60	1.50	1.70	5.20	0.40
500	0%	0	31.71	46.48	16.51	6.80	44.40	38.00	10.70
500	25%	30	50.16	62.24	41.81	7.20	33.10	34.80	8.00
500	50%	50.97	62.89	73.25	59.81	7.20	25.70	32.70	5.90
500	75%	76.07	75.77	85.9	79.48	7.20	17.60	29.50	3.50
500	100%	99.99	87.5	96.87	99.61	6.90	8.70	22.10	2.40
800	0%	0	31.46	43.90	23.96	12.10	78.10	68.30	14.00
800	25%	27.81	45.78	59.35	41.72	12.50	63.10	63.50	14.30
800	50%	48.27	59.00	70.63	57.72	12.50	49.20	59.20	11.50
800	75%	79.44	77.30	87.04	83.43	12.90	29.10	52.90	5.10
800	100%	99.99	87.5	96.87	99.61	12.50	15.70	40.40	4.30
1500	0%	0	25.95	41.55	17.69	20.00	144.40	115.30	32.90
1500	25%	27.98	43.34	57.99	41.05	20.40	107.30	105.90	24.70
1500	50%	48.21	57.03	69.41	58.68	21.30	84.30	98.90	18.50
1500	75%	77.08	75.32	85.40	80.63	21.50	51.60	89.20	10.80
1500	100%	99.99	87.50	96.87	99.61	21.30	26.70	67.20	7.20

Table 4.2: Compression ratio and compression and decompression times for integer datatype.

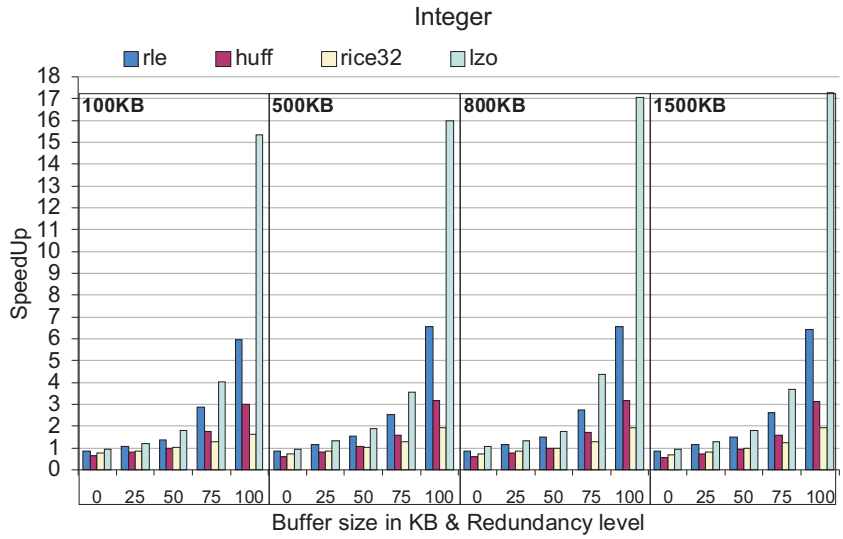


Figure 4.8: Speedup for integer data with different buffer sizes and redundancy levels.

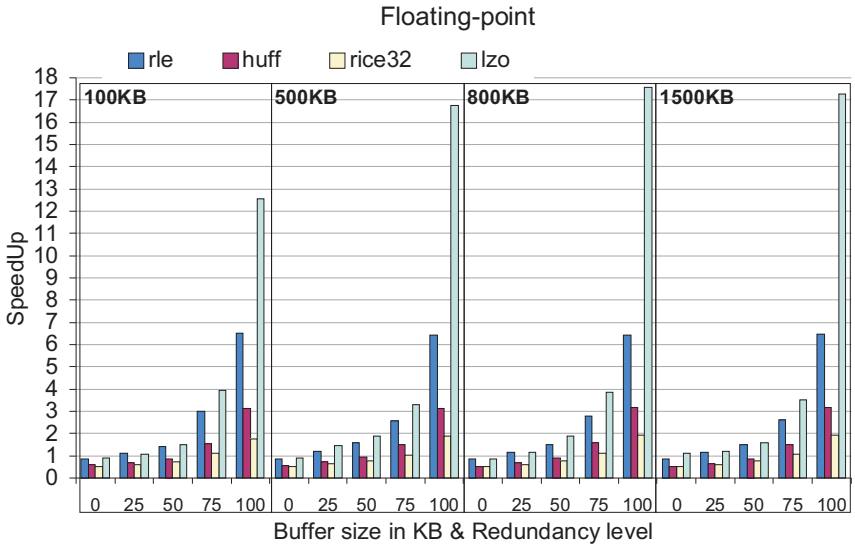
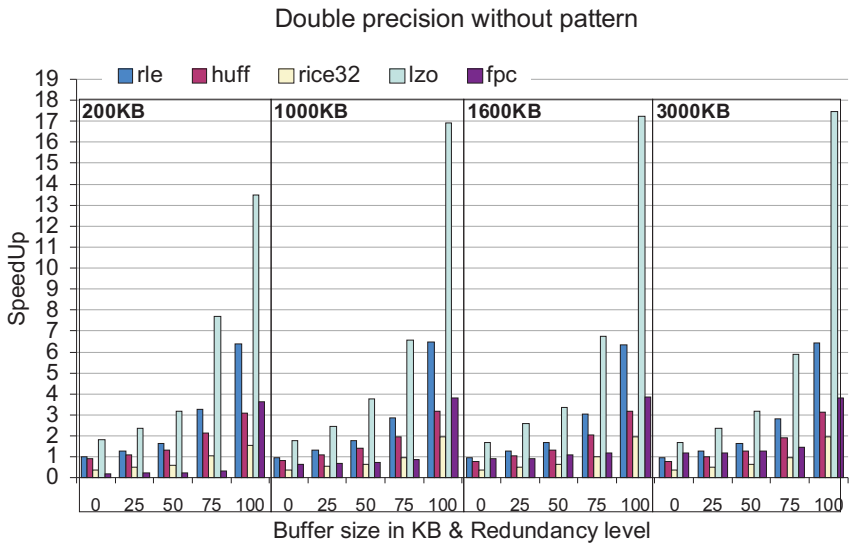


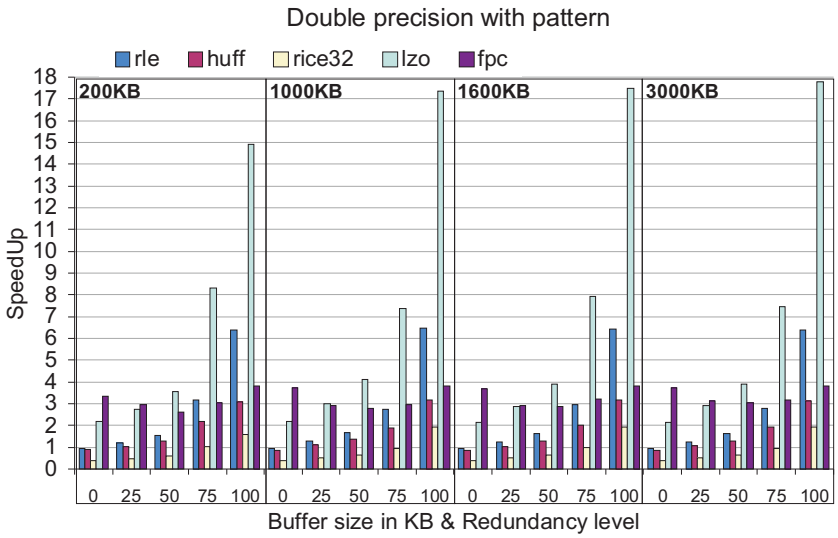
Figure 4.9: Speedup for floating-point data with different buffer sizes and redundancy levels.

traces with a pattern and redundancy level between 0% and 50%, the best results are obtained using FPC. In other cases, the best option is the LZO algorithm.

Finally, *Compression-behavior* builds a list with the algorithms selected for each datatype: the LZO algorithm for integer and floating-point data, and the LZO and FPC for double-precision data.



(a)



(b)

Figure 4.10: Speedup for double data: (a) without pattern and (b) with pattern.

4.5.2 Relationship between compression and decompression time

When a sender process estimates the speedup (equation 4.6) within MPI by using the RAS strategy, it needs the time to decompress the data, and this information is not available in the sender process. Decompression time has to be estimated using as input parameters the compression time, the compression factor, and the message datatype.

Table 4.3 called *Relationship_ Table*, was obtained by measuring the time to compress and decompress for the developed synthetic datasets, and studying the relationship between them. Empirically, we have determined that the datatype factor influence is very low, and that the most important property to predict the decompression time is the compression factor. For example, if we want to estimate the time to decompress a buffer with a 10% *compression_ ratio* by using RLE algorithm, we only have to multiply 0.29 by the time needed to compress the buffer, as equation 4.7 shows. Therefore, by using Table 4.3, *compression_ ratio* and the *T_ compress*, the decompression time of a buffer can be quickly estimated with minimum overhead.

$$T_{decompress} = Relationship\_Table[Algorithm, Compression\_ratio] * T_{compress} \quad (4.7)$$

4.6 Runtime Adaptive Compression Strategy Pseudocode

The *Runtime Adaptive Compression* (RAS) strategy allows selecting dynamically the most appropriated compression algorithm to be used per message exchange, and the size threshold from which a benefit is achieved by using in memory data compression. If no benefit is achieved, compression is not used. As does the *Runtime Compression* strategy explained in Section 4.2, the RAS strategy applies runtime compression (only when it is worth it) to all type of communications and messages: collective, point-to-point, blocking, non-blocking, contiguous and non-contiguous. Therefore, we propose modifying the ADI layer to include the RAS strategy in order to send and receive communications in blocking and also in non-blocking mode. As occurs in the *Runtime Compression* strategy, RAS also distinguishes between blocking or non-blocking communications, and between contiguous or non-contiguous messages, and applies the RAS strategy in each case.

As follows we illustrate in Figure 4.11 the pseudocode of Send_Message_Blocking algorithm for the RAS strategy that corresponds to how the strategy works with blocking sends by using contiguous and non-contiguous messages.

The pseudocode of Send_Message_blocking algorithm for the RAS strategy (illustrated in Figure 4.11) has the follows steps: Firs , the algorithm checks if the

Compression_ratio	RLE	HUFF	RICE	LZO	FPC
0%-25%	0.29	1.33	0.59	0.19	0.99
26%-50%	0.30	1.24	0.57	0.21	0.95
51%-75%	0.25	1.04	0.60	0.23	0.96
76%-100%	0.16	0.45	0.48	1.11	0.93

Table 4.3: Relationship between compression and decompression time for each algorithm.

```
 begin Send_Message_Blocking pseudocode {

input:  buff buffer where is located the data to send
        *dtype_ptr  pointer of data type of data
         dest_rank  identification of process receiver
         src_rank  identification of process sender
         count  Number of element to send

 CHECK IF DATA ARE CONTIGUOUS OR NOT
L1    conitg_size ← Get_datatype_size(dtype_ptr)

 APPLY RAS STRATEGY FOR CONTIGUOUS BUFFER > 2048 BYTES
L2    if(contig_size > 0)
L3          len_buffer = contig_size * count
L4          if(len > 2048)
L5              RAS(buff, len, dtype_ptr, src_rank, dest_rank, blocking_mode)

 SEND CONTIGUOS DATA WITHOUT RAS STRATEGY IF BUFFER <= 2048 BYTES
             else
L6               flag_compress = no_compress
L7               buff ← Add_Head(flag_compress, 0, buff)
L8               Send_Message_contiguous(buff, len, src_rank, dest_rank)
            end if

 APPY RAS STRATEGY FOR NON-CONTIGUOUS BUFFER > 2048 BYTES
       else
L9          len_packet, buff_packet ← Pack_Message(buff, len)
L10         if(len_packet > 2048)
L11             RAS(buff, len, dtype_ptr, src_rank, dest_rank, blocking_mode)

 SEND NON-CONTIGUOUS DATA WITHOUT RAS STRATEGY IF BUFFER <= 2048 BYTES
               else
L12                flag_compress = no_compress
L13                buff_packet ← Add_Head(flag_compress, 0, buff_packet)
L14                Send_Message_contiguous(buff_packet, len_packet, src_rank, dest_rank)
               end if
       end if
 End algorithm}
```

Figure 4.11: Pseudocode of Send message blocking mode in *Runtime Adaptive Compression* strategy (RAS).

message is contiguous or not (*L1* and *L2*). If the message is contiguous, then the second step is to check the size of the message. As a *Runtime Compression* strategy, RAS strategy is not applied when the message size is smaller than 2 KB, because it spends more time performing the compression/decompression than sending the decompressed data. In that case, a flag is added to the message (*Label L6* and *L7*) to indicate to the receive process that it has not decompressed the message. Finally, the message is sent (*Label L8*) in blocking mode. However, if the message is larger than 2 KB, then the adaptive strategy is performed *(L5)*. This means that RAS not only decides to compress or not the message but that it also decides which is the best compression algorithm per message. Pseudocode of Figure 4.12 shows the flow diagram corresponding to this step. Otherwise, if the message is non-contiguous, then it packs the message in a new buffer contiguous (*Label L9*), and performs the

same steps as depicted before (since*L10* to *L14*), which means, checking the message size, and if the message is larger than 2 KB (*Label 10*), the RAS strategy is applied (*Label 11*). But, if the message is smaller than or equal to 2 KB, it is sent without compression (*Labels L12, L13* and *L14*).

RAS also sends the message in blocking or not blocking mode, depending on the communication type. Therefore, the algorithm of Send_Message_Nonblocking is similar to this one, but the messages are sent in non-blocking mode instead of blocking the inside the RAS strategy and also in*Labels L8* and *L14*.

As we explained before, the RAS strategy consists in a applying compression only when it obtains a benefit and with the best compression algorithm per message. Therefore, the receive decompression algorithms (blocking and non-blocking) are exactly the same as the algorithms explained in Section 4.2, because the RAS strategy is only applied to the sending process. Thus, the receive process has to study the head of message to know if it has to decompress the message or not, and which is the algorithm to use for decompressing. If the message is received in blocking mode (for example `MPI_Recv)`, the decompression steps are performed when the received call is finished. However, if the message is received in non-blocking mode (for example `MPI_Irecv`), then the decompression steps are performed after the wait call.

As follows, we depict the internal steps of RAS strategy corresponding to the *Labels L5* and *L11* in the pseudocode of Figure 4.11. Initially, we proposed applying compression by analyzing the content per message (by means of calculating the message entropy) in RAS strategy, and we have observed that the overhead introduced is very high. So we studied other ways to make this decision more efficiently. We found that message length and datatype are the major characteristics that can be used to consider whether to compress or not, and which compression algorithm should be used.

Taking into account the compression algorithm solutions, we distinguish three kinds of datatypes to cover the most usual standard datatypes in parallel applications, and to allow the usage of application-defined datatypes:

- Integer and float-point datatypes. Based on the results of *Compression-behavior* described in Section 4.5, for these datatypes, we apply the algorithm LZO.
- Double-point datatype. In order to select the best algorithm for this datatype, it is necessary to study if the data has a pattern or not. The algorithms selected by *Compression-behavior* are LZO and FPC.
- Others datatypes. These datatypes are built by the application. There are four candidate algorithms for these datatypes: LZO, RLE, RICE, and HUFFMAN.

RAS analyzes the three kinds of datatypes separately. Each datatype has its own metadata variables in order to take the best decision: *num_ messages*, *length_ yes _ compression*, *length_ no_ compression*, *num_ uncompress*. These variables are detailed below. When a process sends a message, RAS first checks the message datatype, and then applies the runtime strategy associated with that datatype. This means

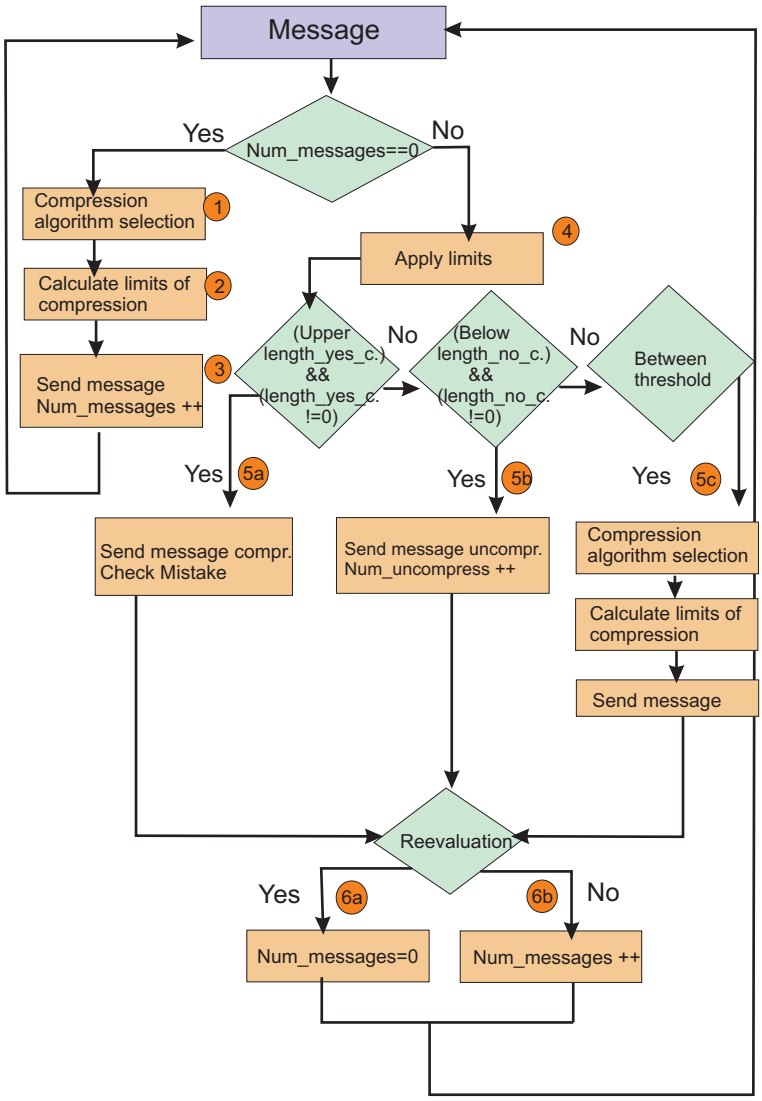


Figure 4.12: Flow diagram of the runtime adaptive strategy.

that, for example, for integer datatypes, RAS can decide to compress data from 4 KB by using LZO algorithm, while for double datatype, the runtime can decide not to apply compression, and for others datatypes it has decided to apply RLE algorithm from messages larger than 8 KB. These decisions take into account the network and compression behavior heuristics fed to the system.

The RAS (Labels L5 and L11 in pseudocode of Figure 4.11) decision process (depicted in Figure 4.12) consists of the following steps:

1.- Selecting the best compression algorithm. This step is performed only when the value of *num_ messages* is equal to zero. This means that if it is the first message, or we have restarted the study, then, the process selects the best compressor for the considered datatype following procedure: First, the process compresses the message with the compressors preselected by *Compression-behavior*. Next, it compares the speedup (calculated using equation 4.6) of

each one of them. Notice that, in order to calculate the speedup we have to apply equation 4.7 to estimate the $T_{decompress}$ and equation 4.5 to estimate *time_to_send_message* uncompressed and compressed. These values are provided by *Network-behavior*. Finally, the process selects the compressor expected to achieve a higher speedup.

2.- Finding the minimum size of the message from which the performance improves. If the highest speedup computed in step 1 is less than one, then the process sets the size of message as *length_no_compression*. Otherwise, if it is greater than one, it then sets the length as *length_yes_compression*.

3.- Sending the message. In this step, the message is sent (compressed or uncompressed), and the process updates the number of messages sent.

4.- For subsequent messages with the same datatype, the process compares the message size with the *length_no_compression* and *length_yes_compression*. If the size of the message is higher than *length_yes_compression*, then the message is sent compressed, and if it is less than *length_no_compression* , then the message is sent uncompressed. On the other hand, if the size of the message is between both thresholds, it is necessary to repeat the evaluation: compressing the message, checking the speedup, and updating the value of one of the two thresholds. Note that this produces more accurate threshold values, that is, the gap between the threshold values is reduced after each evaluation.

5.- Once the message is sent, and if the decision is to compress, the process checks if a *mistake* is produced: This means that it checks whether the *compression_ratio* is less than one. In that case, the process studies if it has to restart the study, depending the number of mistakes and their interval of time.

6.- In the other case, if the decision was uncompressed, the process updates the number of messages that have been sent uncompressed. When the number is higher than a *reevaluation threshold*, then the evaluation is restarted. The reevaluation threshold is computed during MPI installation. For this study, it is set to 1000.

For all messages sending in RAS strategy, a header is included to inform the receiver process if it has to decompress the message or not, and which algorithm must be used.

One of the main characteristics of RAS is that it can adapt itself to the applications' behavior at runtime. This strategy learns from previous messages which is the compression algorithm to be used and the size from which it obtains a benefit by compressing data. For example, heuristics might say that LZO is better for integers, but for some applications using this datatype it could be better to use another compressor, as RLE, depending on message data. The same situation might arise with double point floats if patterns are not present.

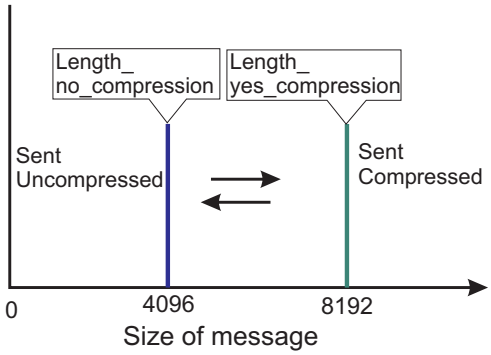
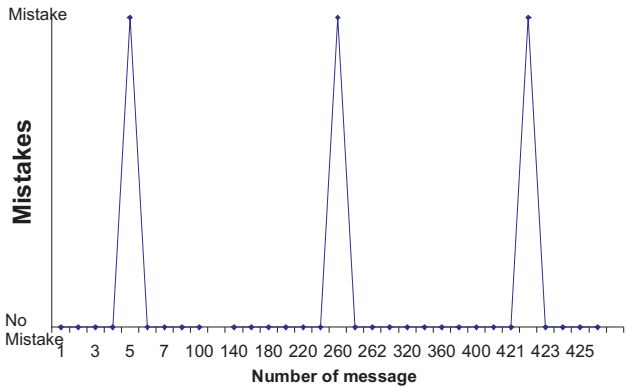


Figure 4.13: Finding the message size from which we achieve a benefit compressing data.

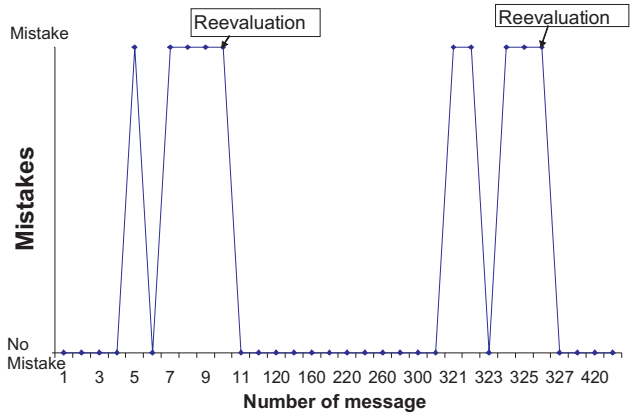
The initial values for *length_no_compression* and *length_yes_compression* are zero. Their values are updated for the first message, where steps 1, 2 and 3 of Figure 4.12 are applied. If the compression speedup and compression ratio are greater than one, *length_yes_compression* is set to the message length. Otherwise, *length_no _compression* is set instead. This procedure is subsequently applied to the following messages with length between these two thresholds, dynamically modifying their values (see step 5c).

If the data of the application messages are very similar, once the compression limits are calculated (*length_no_compression* and *length_yes_compression*), there will be no mistakes. For example, if we consider the situation shown in Figure 4.13, when a message is larger than *length_yes_compression* (8192 bytes), then the message is sent compressed, and if it is less than *length _no_compression* (4096 bytes), then the message is sent uncompressed. When the size of the message is between both marks, the process has to compress the message and check the speedup. Therefore, the process will update the value of one of the two marks. During the execution of an application, both marks (*length_no_compression* and *length_yes_compression*) will approach, until they converge in the same value.

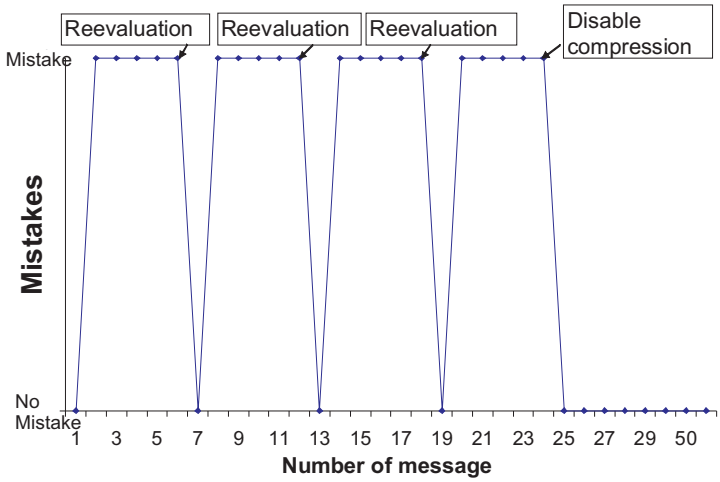
For other cases, if the message contents or sizes change significatively, RAS can make *mistakes* by compressing messages without getting any speedup. The RAS strategy detects a mistake when it has compressed a message because its size is greater (or equal to) than *length_yes_compression* and its compression ratio is smaller than one. Because of that, we have to account for the number of mistakes, and their interval, in order to start the reevaluation or not. For example, if during the execution of an application, a process makes some mistakes, but in very long intervals (as shown in Figure 4.14(a)), it is better not to reevaluate because the cost would be too high, and most of the time the decisions are correct. However, if the errors are made frequently (small intervals), the correct decision is to reevaluate, as shown in Figure 4.14(b). Finally, if during a short period of time, a process has to be reevaluated many times, then the best decision is to disable the compression and the adaptive module, as Figure 4.14(c) shows, as the cost to reevaluate would be greater than the benefit of sending the data compressed.



(a)



(b)



(c)

Figure 4.14: Different cases of reevaluation: (a) No reevaluation, (b) Reevaluation and (c) Disable adaptive module.

4.7 Guided Strategy

Note that the RAS strategy is a general solution, and it assumes that the application behavior is not known. However, the evaluation creates an overhead that can be mitigated if the application behavior is known in advance. For this reason, we have developed the *Guided strategy* (GS) to take very effective decisions with a minimum cost. GS strategy is accomplished by storing all messages from the application during the first execution. Next, all the messages stored for each process are compressed and decompressed with the compressors selected, obtaining speedup estimations by sending compressed messages. Using this information, each process creates a rule for each message indicating the algorithm that obtains the highest speedup. If speedup is lower than one, no compression is chosen. This information is stored in a *Decision rules* file per process, so that for subsequent executions of the same application, with the same input parameters, the GS strategy only has to apply the rule for each message before sending a message, thus avoiding the overhead of the RAS.

The Guided Strategy is based on each process appliying the decisions (on whether to compress and which algorithm), which has been off-line calculated before send a message. We have developed two modules to make off-line application profiling. One module, linked with the application, stores all messages per process in files called *Trace files* (one *Trace file* per process) during the first execution of the application. Also, each process stores in its *Trace file* the main characteristics of each message (datatype, sender and receiver rank, length of buffer, message tag, communicator for the operation, numeric context id and type of communication).

When the execution is over, another module compresses and uncompresses all messages kept in all *Trace files*, with the LZO, RLE, Huffman, Rice and FPC compression algorithms. Moreover, the module calculates the speedup and compression ratios for each compression algorithm. Based on the results obtained for each message, the module creates a rule for every message. These rules contain the information about whether to compress or not, and which compression algorithm to be used. This information is stored in a *Decision rules* file per process.

We have taken into account that eventually the messages can arrive in different order, despite using the same input parameters. For this reason, we store in the *Decision rules* file the rule of each message with an identifier associated with the message. The identifier is calculated by using a hash based on the message characteristics of the message (datatype, rank of sender, etc.) and it is kept in the trace file.

Decision rules are used to guide GS compression to take very effective decisions with a minimum cost. Using this information, for subsequent executions of the same application with the same input parameters, GS applies the rule for each message before sending it, thus avoiding the overhead of the adaptive run-time approach.

For long running applications the use of trace files could generate huge data volumes that would limit the applicability of this approach. However, most of these applications are iterative, and the same communication pattern is repeated inside a loop (usually related to the simulation time, where each iteration is a time step).

```
 begin Send_Message_Blocking pseudocode {

input:  buff buffer where is located the data to send
        *dtype_ptr  pointer of data type of data
         dest_rank  identification of process receiver
         src_rank  identification of process sender
         count  Number of element to send

 CHECK IF DATA ARE CONTIGUOUS OR NOT
L1      conitg_size ← Get_datatype_size(dtype_ptr)

 GUIDED STRATEGY FOR CONTIGUOUS BUFFER
L2      if(contig_size > 0)
L3         len_buffer = contig_size * count
L4         GS(buff, len, dtype_ptr, src_rank, dest_rank, blocking_mode)

 GUIDED STRATEGY FOR NON-CONTIGUOUS BUFFER

       else

L5           len_packet, buff_packet ← Pack_Message(buff, len)

L6           GS(buff, len, dtype_ptr, src_rank, dest_rank, blocking_mode)

        end if

    End algorithm}
```

Figure 4.15: Send_Message_Blocking pseudocode for Guided Strategy (GS).

In these cases, a cycle detection algorithm would detect the communications related to each iteration. As future work, we want to modify the Guided Strategy to create the Decision files on the fly, based in a new cycle detection algorithm.

As *Runtime Compression* and RAS strategies, we propose modify the ADI layer to include GS strategy in order to compress send/receive communications with blocking/non-blocking modes by using contiguous/non-contiguous datatypes. As follows we illustrate the pseudocode in Figure 4.15 corresponding to Send_Message_Blocking mode in GS strategy.

The pseudocode illustrated in Figure 4.15 of Send_Message_Blocking algorithm for GS strategy has the next steps: Firstly, the algorithm checks if the message is contiguous or not (*L1 and L2*). If the message to send is contiguous, then GS strategy is applied (*Label L3*). Otherwise, if the message is non-contiguous, then it packs the message in a new buffer contiguous (*Label L5*), and applies GS strategy (*Label L6*). The GS strategy is applied to all messages, includes those who are smaller than 2 KB. This strategy stores all messages in the first execution, and in the subsequent executions, it reads the decision rule per message. Note that, this pseudocode is the same when GS is applied in the first executions, or in the others. Internally, the strategy detects if it has to store the messages or it has to apply the decision rules, as shown Flow Diagram 4.16.

GS strategy is applied into blocking and non-blocking sends. Therefore, the algorithm of Send_Message_Nonblocking is similar than the explained before, but the messages are sent in non-blocking mode instead of blocking inside GS strategy.

As we explained before, GS strategy consists to apply the decision rule (on

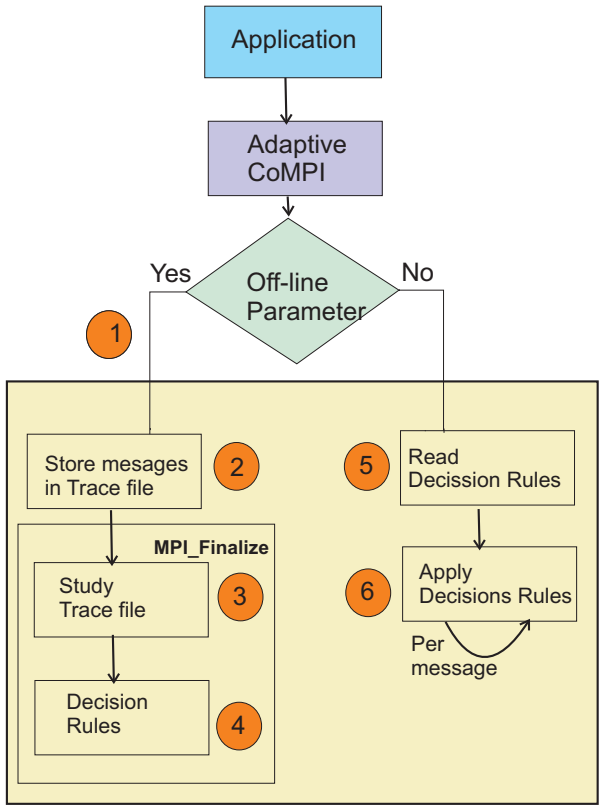


Figure 4.16: Flow diagram of the GS strategy.

whether to compress and which algorithm) per message. Thus, the GS strategy only is applied to the sender process. So, the receive decompression algorithms (blocking and non-blocking) are exactly the same as the algorithms explained in Section 4.2.

Below, we detail the steps of GS (see Figure 4.16) corresponding to labels *L3 and L6* in Figure 4.15:

1.- Executing the application with an input parameter to indicate to the GS strategy that it has to start to trace the application.

2.- Each process stores the messages sent to other processes in a file called the *Trace File*.

3.- At the end of the program execution, each process evaluates its *Trace file* in the `MPI_Finalize` function. This means that each message stored in *Trace files* is compressed and decompressed with all the compression algorithms. Subsequently, the speedups are calculated.

4.- As a result of the study, each process writes a zero to a file if all algorithms achieve a speedup no higher than one, or the identifier of the algorithm with the highest achieved speedup. Each process names this file with the execution parameters and with its rank, but in this thesis, we generally refer to the name as *Decision file*.

5.- In the next execution with the same parameters, the processes read their *Decision files* only once, and store the information in a memory table.

6.- Finally, each process, before sending a message, reads the table and applies the decision previously computed.

As RAS, the GS strategy also adds a head in all transferred messages in order to inform the receiver process whether it has to decompress the message or not and which algorithm must be used.

To calculate the speedup per message, equation 4.6 is applied. For each message, compression and decompression time are known, but we need to estimate the time to send the compressed and decompressed data. With this purpose, the information provided by *Network-behavior* in the network transmission time is employed (Table 4.1). Then, the time to send the data compressed and decompressed can be estimated. Finally, the speedups per message are calculated by using different compressors.

Notice that to create the *Trace files*, the application must be run entirely to log its behavior, which might require high storage capacity depending on the number of communications performed by each process of an application. However, after postprocessing, the size of the *decision rules* files can be comparatively very small. Table 4.4 shows application logging time, the size of *Trace File*, and the size of *Decision File* for each process of several applications used to evaluate our system. For example, BISP3D is a semiconductor simulation application. As shown in the table, its execution time with 16 processes takes 152 seconds, and the *Trace file* size per message is 801 MB, but its decision files are very reduced (only a few KB).

The GS strategy is especially useful for applications that are executed many times on the same system and with the same conditions, or for applications with a short execution time and relatively low number of communications. However, both conditions are very frequent in many community or service clusters, where the set of applications is relatively small and optimizing the execution time is important. Performance enhancement achieved by applying GS instead of RAS for an application are studied in the Evaluation Section 6.5.2.

Application	Execution_Time (sec)	Trace_File (MB)	Decision_File (KB)
BISP3D	152	801	2
PSRG	6.6	17.4	1.6
STEM	37	240	12
NAS-IS	162	352	0.5
NAS-LU	206	360	2

Table 4.4: The Trace and Decision File size per process with different applications.

4.8 Adaptive-CoMPI Technique

We have choosen the *MPICH* implementation to intregrate both strategies, and we have called it *Adaptive-CoMPI* technique. *Adaptive-CoMPI* [FCS+10a] can fit any particular application because its implementation is transparent for users. The *Adaptive-CoMPI* technique integrates different compression algorithms and GS and RAS strategies are included in order to decide whether to apply compression or not:

- Run Time Strategy (RAS): This strategy learns in run-time from previous messages to choose the compression algorithm to be used and the minimum message size that allows a benefit from sending the message compressed to be obtained.

- Guided Strategy (GS): This strategy executes firstly an off-line application profiling. Secondly, each process studies which is the best choice (applying compression or not, and which algorithm to use) for each message transferred. Finally, all processes keep these decisions in a *Decision Rules* file per process. In consecutive executions of the same application (with the same input parameters), the *Run-Time Adaptive CoMPI* module reads from *Decision Rules* files the effective decision to take for each message with minimum cost for the best compression algorithm.

The selection of the strategy to be applied in *Adaptive-CoMPI*, depends on the user. By default, *Adaptive-CoMPI* always chooses RAS, but the user can indicate the selection of GS by an MPI hint.

The goal of the *Adaptive-CoMPI* technique, independent the strategy selected, is to use the most appropriate compressor (including the null compressor) in order to maximize the speedup per message and to reduce the applications execution time.

4.8.1 Message Compression

Different compression algorithms are used depending on the specific characteristics of each communication in RAS and GS strategies. Also, our strategies allows using (or not) compression per message, depending on the theoretical speedup per message calculated by using the network and compression heuristics. All compression algorithms have been included in a single library called *Compression-Library*. To include more compression algorithms, we only have to replace this library with a newer version. Therefore, the *Adaptive-CoMPI* can be easily updated to include new compression algorithms. Currently, the compression library includes: RLE, Huffman, Rice, LZO, and FPC.

As explained in Sections 4.3 and 4.7, for both strategies (GS and RAS), a header is included in the exchanged messages in order to inform the receiver process whether it has to decompress the message or not, and which algorithm must be used. The header contains the following information:

- Compression used or not: a byte that indicates if the message is compressed or not.
- Compression algorithm: a byte that indicates which compression algorithm has been selected.
- Original length: an integer that represents the original length of data.

As described in Sections 4.6 and 4.7, depending on the chosen stategy, the selection of the compression algorithm and the decision to send the message compressed or not, are taken in different ways. But once made, the messages are compressed and decompressed in the same way for the RAS and GS strategies as shown in the pseudocode of Figure 4.17. The same applies for the *Runtime Compression* strategy explained in Section 4.2. These algorithms (Compression and Decompression messages) are included inside the *Compression-Library*.

The stages of compression and decompression algorithms are detailed below:

- Compression_Message pseudocode:
  1.- Data compression (Label L1): Compressing data and checking the size of the compressed data. If the size of compressed data is greater than the original data, the original message is sent (Label L5).
  2.- Header inclusion (Label L2): Adding a header to the message in order to notify the receiver whether the message has to be decompressed and which decompression algorithm has to be used.
- Decompression_Message pseudocode steps:
  1.- Header checking (Label L1): Checking the header in order to know whether the message has to be decompressed or not and which algorithm has to be employed.
  2.- Data decompression (Label L2): Applying the decompression algorithm indicated by the sender.

4.8.2 Integration of network and compression behavior in *Adaptive-CoMPI*

The *Compression-behavior* test is executed on the *Adaptive-CoMPI* installation, so it is transparent to users. This benchmark generates an output file with the results of the compression algorithms for each datatype, and the relation between the compression and decompression time for each compressor. *Adaptive-CoMPI* reads this file in the `MPI_Init` function, and its contents are stored in memory to be used during the application execution as heuristics inputs for RAS.

```
 begin Compression_Message pseudocode {

input:  buff  buffer to compress
        len size in bytes of buffer
        algorithm algorithm to compress the message
        flag indicate if the buffer it is compressed or not

output: buff_compress buffer compressed
        len_compress len of buffer compressed

 CHECK  COMPRESSION ALGORITHM AND COMPRESS THE BUFFER

L1    swich (algorithm :)
                LZO : len_compress, buff_compress ← Compression_with_lzo(buff, len)
                RLE : len_compress, buff_compress ← Compression_with_rle(buff, len)
                FPC : len_compress, buff_compress ← Compression_with_fpc(buff, len)
                RICE : len_compress, buff_compress ← Compression_with_rice(buff, len)
                HUFFMAN : len_compress, buff_compress ← compression_with_huffman(buff, len)
        end swich

 ADD HEADER TO BUFFER
L2     if len_compress < len
L3        flag = yes_compress
L4        buff_compressed ← Add_Head(flag, algorithm, buff_compress, len)
       else
L5           flag = no_compress
L6           buff ← Add_Head(flag, 0, buff, len)
       end if

   End algorithm}

.....


 begin Decompress_Message pseudocode {

input:  buff_compress  buffer compressed

output: buff buffer to decompress

 STUDY HEADER OF BUFFER

L1      algorithm, len_original, flag_compression ← study_header(buff_compress)
        if (flag_compression == yes_compression)
 DECOMPRESS BUFFER
L2         swich (algorithm :)
                LZO : buff ← Decompression_with_lzo(buff_compress, len_original)
                RLE : buff ← Decompression_with_rle(buff_compress, len_original)
                FPC : buff ← Decompression_with_fpc(buff_compress, len_original)
                RICE : buff ← Decompression_with_rice(buff_compress, len_original)
                HUFFMAN : buff ← Decompression_with_huffman(buff_compress, len_original)
            end swich
        end if
 End algorithm}
```

Figure 4.17: Compression and Decompression algorithms pseudocode

Network-behavior test can be executed in two different execution stages:

- During the installation of *Adaptive-CoMPI*. This is the default option, and it is executed at the same time as the *Compression-behavior*. This benchmark needs a list of machines in order to model the cluster network. The *Network-behavior* generates an output file with the transmission time per byte (Table 4.1). The `MPI_Init` function is also responsible for storing this information in memory. The advantage of this approach is that it is transparent to the user, and it produces no overhead for the application. In contrast, if there is a change in the network technology, the benchmark would not be able to detect it until a new installation of *Adaptive-CoMPI* is executed.

- In *MPI_ Init* function: The *Network-behavior* is performed automatically at the very beginning of the MPI application (in `MPI_Init` function) execution in order to calibrate the communication time between every pair of nodes assigned by MPI run (list of machines to be used). This option is adaptive: whatever network technology is installed, the benchmark capture the new situation. But this option produces overhead when it is executed by the application. This option to be used can be selected by an MPI hint.

The evaluations done in our experiments, we have executed the *Network-behavior* during the installation of *Adaptive-CoMPI*.

4.9 Summary

In this chapter we have proposed reducing the volume of MPI communication applying lossless compression. We have developed three strategies in order to compress messages in runtime. The first strategy, called *Runtime Compression*, compresses all transferred messages of an application with the same compression algorithm indicated by the user. In addition, all the messages are compressed without the possibility deactivating compression.

The second strategy developed, called *Runtime Adaptive Compression* (or RAS), is a improvement over the former strategy. RAS is able to select in runtime the most appropriate compression algorithm in order to maximize the speedup per message. The strategy can dynamically adapt to the application behavior, learning from the previous communication history, and taking into account the specific communication characteristics (datatype, message size, etc), the platform specifications (network latency and bandwidth), the compression technique performance, and a threshold message size. Furthermore, it is able to activate and deactivate itself in runtime when the compression is not worth it.

The third strategy proposed, called the *Guided Strategy* (or GS), avoids the RAS overhead by reducing the decision time. The GS is based on each process applies the decisions (on whether to compress and which algorithm), which has been off-line calculated before sending a message.

We have integrated RAS and GS strategies in MPI generating a modified implementation called *Adaptive-CoMPI*. *Adaptive-CoMPI* can fit any particular application because its implementation is transparent for users. *Adaptive-CoMPI* technique integrates different lossless compression algorithms.

Moreover, in this chapter, we have developed two models (network and compression behavior) in order to study the compression algorithms and network characteristics. First, using synthetic traces, we analyzed the performance (both in terms of compression ratio and execution time) of each compression technique. Different factors (type of data, buffer sizes, redundancy levels and data patterns) were considered, generating an output file with the best compression algorithms for each datatype. Second, we have measured the latency and bandwidth in our cluster by using MPI primitives in order to estimate the time to transmit a message. The output of these models are two heuristic files (network and compression) used by *Adaptive-CoMPI* to decide, the compression algorithm to be used for each message.

Chapter 5

Dynamic-CoMPI: Dynamic Techniques for MPI

5.1 Introduction

In this chapter, we propose an optimization of the MPI library, called *Dynamic-CoMPI*, which combines the techniques presented in Chapters 3 and 4 in the same MPI implementation. By means of this joint approach is is possible to reduce the impact of communications and non-contiguous I/O requests in parallel applications. This technique employs different strategies that are independent of the application characteristics and they can be transparently applied at run-time without introducing modification in the application structure.

The first technique, called *LA_ TwoPhase I/O*, reduces the number of communicaiton involded in Two-Phase collective I/O operation, by using a new aggregation pattern. This technique employs the Linear Assignment Problem (LAP) to find an optimal I/O data communication schedule.

The second technique, called *Adaptive-CoMPI*, uses message compression, in order to reduce the communication time of individual messages and to efficiently exploit the available bandwidth of the overall system. *Adaptive-CoMPI* allows selecting dynamically the most appropriated compression algorithm to be used per message exchange. In addition, it is able to turn itself on and off. The compression is applied to all communications of MPI, including the ones performed in I/O subsystem. Therefore, the I/O system is benefitting from both optimizations(*LA_ TwoPhase I/O* and *Adaptive-CoMPI*), because on the one hand the number of communications is reduced with the new pattern of aggregators, and on the other, the volume of communications is also reduced by compression. This means that the I/O time is reduced by combining both techniques.

In order to develop a joint implementation of both approaches, we propose integrating both techniques in MPI internal structure with the least overhead possible, and in a transparent way for users and applications. Therefore, our proposal can be applied at run-time without introducing modification in the application structure or source code. The following sections depict the architecture and internal structure of *Dynamic-CoMPI*.

5.2 Dynamic-CoMPI Architecture

In this section we describe the *Dynamic-CoMPI* architecture, designed as an extended MPI run-time implementation with new functionalities. *Dynamic-CoMPI* combines the main strategies proposed in this Ph.D. tesis:

- Data compression to reduce the volume of communications.
- New aggregation patterns to reduce the number of communications performed in I/O operations.

Figure 5.1 shows the internal *Dynamic-CoMPI* architecture. The most important layers associated with communication and I/O system are:

- The Application Programmer Interface (API): The interface between the programmer and Abstract Device Interface (ADI). It uses the functionalities implemented in ADI layer for sending and receiving information.
- The Abstract Device Interface (ADI): Controls the data flow between API and hardware. It specifies whether the message is sent or received, handles the pending message queues, and contains the message passing protocols.
- The Abstract Device I/O interface (ADIO): It consists of a small set of basic functions for parallel I/O, including collective I/O operations as *Two-Phase I/O*.

Our modifications cover two different elements of the MPI internal structure. On one hand, we have modified the ADIO layer to replace the default I/O aggregator pattern in *Two-Phase I/O* by a dynamic I/O aggregator pattern. It decides how to distribute I/O aggregators among all processes according to the local data that each process contains. The *LA-Two-Phase I/O* technique is implemented in the ADIO layer.

On the other hand, we have modified the ADI layer in order to include compression facilities to all MPI communications (point-to-point and collective). For this reason, the *Adaptive-CoMPI* technique is located in ADI layer. Compression is applied in all MPI communications, including the ones that are performed for I/O. Therefore, the I/O subsystem benefits from both optimizations because the number of communications is also reduced with the new I/O aggregators pattern,

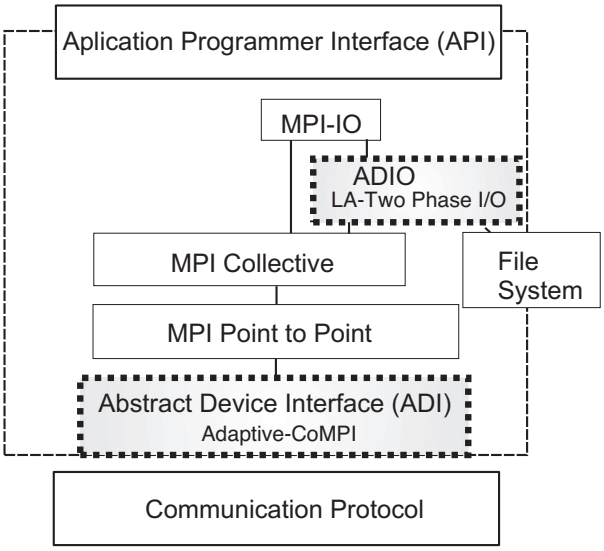


Figure 5.1: Internal *Dynamic-CoMPI* architecture.

and the volume of communications is reduced by using compression. The evaluation Section 6.6 shows a detailed performance evaluation of the technique. Results show that the I/O operation time is reduced considerably by combining both techniques.

We have implemented *Dynamic-CoMPI* into the *MPICH* [GL97] implementation of MPI, developed jointly by Argonne National Laboratory and Mississippi State University.

5.3 Adaptive-CoMPI Modification

As we explained before, *Dynamic-CoMPI* uses data compression to reduce the volume of communications. Less volume of data to be transferred means reducing execution time and enhancing scalability of applications. In this work, we have implemented two different compression techniques: *CoMPI* and *Adaptive-CoMPI* explained in Sections 4.2 and 4.8 respectively. We have chosen the *Adaptive-CoMPI* compression technique to implement *Dynamic-CoMPI* for the following advantages:

- *Adaptive-CoMPI* has integrated two different compression strategies in order to decide whether to apply compression or not:
  - Runtime Adaptive Compression (RAS): This strategy learns in run-time from previous messages to choose the compression algorithm to be used and the minimum message size that allows a benefit from sending the message compressed to be obtained.
  - Guided Strategy (GS): This strategy is based on each process applying the decisions (on whether to compress and which algorithm) which have been calculated off-line before sending each message.

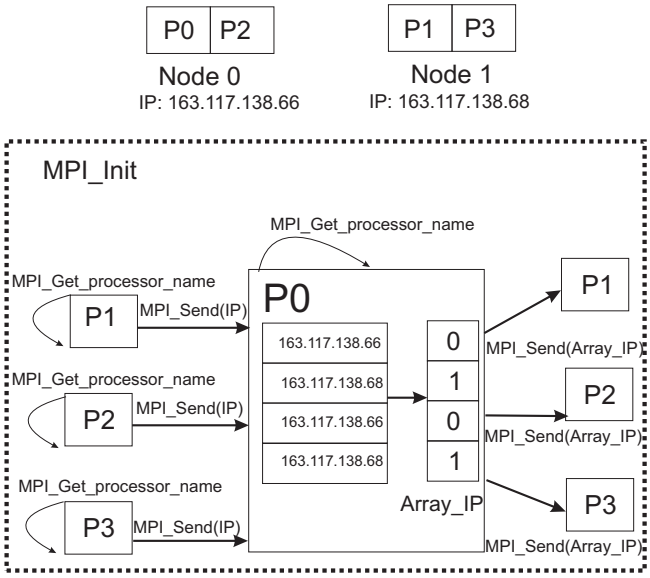


Figure 5.2: Building *Array_IP* to detect if two processes are on the same node.

- The selection of the strategy to be applied in *Adaptive-CoMPI* depends on the user. By default, *Adaptive-CoMPI* always chooses the RAS, but the user can indicate the selection of GS by an MPI hint. Note that the*Adaptive-CoMPI* technique, independent of the strategy selected, uses the most appropriate compressor (including the null compressor) in order to maximize the speedup per message and to reduce the applications execution time.

The main feature of *RAS* is that it tries to select at run-time the best compression algorithm for each message, and to decide whether it is worth compressing data or not. Obviously, it is not always efficient to compress all messages of an application, thus, in some cases the best choice might be not to compress.

We realized that, if the processes involved in communication are located in the same node, the best choice is to send the data uncompressed. There are many MPI implementations that have shared memory communication support, which naturally need to detect if two processes are on the same node, such as OpenMPI [GFB+04] or Mvapich2 [HSJ+06]. Currently, *Dynamic-CoMPI* is integrated into MPICH 1.2.7.15-NOGM and in this implementation, detection if two processes are on the same node is not implemented. Therefore, we have modified the `MPI_Init` function to allow RAS to verify the location of sender and receiver processes.

As shown in Figure 5.2, process 0 builds an array with as many entries as processes. Each entry contains the IP address of each process. This array is built after all processes consult their name using `MPI_Get_processor_name` function and send this information to process 0. Then, process 0 builds another array called *Array_IP* with an element per process. Process 0 stores the same number for those processes that have the same IP in the *Array_IP*. Finally, it broadcasts the *Array_IP* to all processes, so *Array_IP* becomes a global table, accessible from all layers of *Dynamic-CoMPI*.

Therefore, we have modified the RAS strategy to deactivate the compression when two processess are located in the same node. In addition, we have designed a new function, called `Detection_Location_Processes`, that consults the *Array_IP* table to verify that the receiver process IP is different than or equal to the sender process.

The new pseudocode of RAS for Send_Message_bloclking algorithm 5.3 differs from the original (shown in Figure 4.11), in the following steps: Before sending a message (contiguos or non-contiguos, blocking or non-blocking), RAS checks the location of the processes involved in the communication (*Labels L4* and *L11*), calling to *Detection _location_ Processes* function. If the processes are located in different nodes, then the RAS strategy is applied (*Labels L6* and *L13*), otherwise, the message is sent without compression (*Labels* L7 and *L14*).

5.4 Summary

In this chapter we have proposed the optimization of the MPI library, called *Dynamic-CoMPI*, which combines the techniques explained in Chapters 3 and 4. Our goal is to reduce the overall time of parallel applications combining both techniques. This means reducing the volume of communications with an adaptive compression strategy, and also reducing the number of communications performed in collective operations.

Furthermore, we have modified the *Runtime Adaptive Strategy* explained in Chapter 4, to be able to activate and deactivate itself when the processes involved in communication are located in the same node. The recent popularity of Chip Multiprocessors (CMP) clusters is the reason for this decision. CMP clusters contain more than one core per node. In this kind of clusters, the communication intra-node is very fast because it is a local operation. Thus, it only requires the time of a memory copy operation plus the latency of the network stack, and hence, it is not worthwhile sending the data compressed for intra-node communication.

```
 begin Send_Message_Blocking pseudocode {

input:  buff buffer where is located the data to send
        *dtype_ptr  pointer of data type of data
         dest_rank  identification of process receiver
         src_rank  identification of process sender
         count  Number of element to send

 CHECK IF DATA ARE CONTIGUOUS OR NOT
L1    conitg_size ← Get_datatype_size(dtype_ptr)

 APPLY RAS STRATEGY FOR CONTIGUOUS BUFFER > 2048 BYTES AND PROCESSES LOCATED IN DIFFERENT NODES
L2    if(contig_size > 0)
L3          len_buffer = contig_size * count
L4          location ← Detection_Location_Processes(src_rank, src_dest)
L5          if(len > 2048)AND(location == 1)
L6               RAS(buff, len, dtype_ptr, src_rank, dest_rank, blocking_mode)

 SEND CONTIGUOS DATA WITHOUT RAS IF BUFFER<=2048 BYTES OR PROCESSES LOCATED IN THE SAME NODE
             else
L7               flag_compress = no_compress
L8               buff ← Add_Head(flag_compress, 0, buff)
L9               Send_Message_contiguous(buff, len, src_rank, dest_rank)
            end if

 APPY RAS FOR NON-CONTIGUOUS BUFFER > 2048 BYTES AND PROCESSES LOCATED IN DIFFERENT NODES
       else
L10          len_packet, buff_packet ← Pack_Message(buff, len)
L11          location ← Detection_Location_Processes(src_rank, src_dest)
L12          if(len > 2048)AND(location == 1)
L13               RAS(buff, len, dtype_ptr, src_rank, dest_rank, blocking_mode)

 SEND NON-CONTIGUOUS DATA WITHOUT RAS IF BUFFER<=2048 BYTES OR PROCESSES LOCATED IN THE SAME NODE
              else
L14               flag_compress = no_compress
L15               buff_packet ← Add_Head(flag_compress, 0, buff_packet)
L16               Send_Message_contiguous(buff_packet, len_packet, src_rank, dest_rank)
              end if
       end if
 End algorithm}

.....................................................................................................


 begin Detection_Location_Processes pseudocode {
input:   src_rank  identification of process sender
         dest_rank  identification of process receiver

output:  location  Flag to detect if two processes are located in the same node

L1          location_source = Array_IP[src_rank]
L2          location_destine = Array_IP[dest_rank]
L3          if(location_source ≠ location_destine)
L4            location = 0
            else
L5           location = 1
            end if
 End algorithm}
```

Figure 5.3: New Pseudocode of Send message blocking mode in *Runtime Adaptive Compression* strategy (RAS).

Chapter 6

Experimental Results

6.1 Introduction

The goal of this chapter is to demonstrate the feability of the techniques proposed in this Ph.D. work. To achieve this, we have carried out an experimental evaluation of the techniques presented in Chapters 3, 4 and 5. This chapter is organized in five sections. The first and second sections present the real applications and benchmarks used in this Ph.D. thesis in order to evaluate our proposals.

The third section presents the results of *LA_ TwoPhase I/O* technique presented in Section 3.5. There are three main aspects which we are interested in. First, we make a study of *Two_ Phase I/O* to know which are the algorithm stages with the highest computational cost. Second, we compare if the number of communications performed in *Two_ Phase I/O* is reduced by using the appropriate aggregator pattern instead of the default pattern. The new aggregator pattern is calculated in runtime by applying *Linear Assignment Problem*. Third, we compare the performance of *LA_ TwoPhase I/O* technique with *Two_ Phase I/O* by using different applications and architectures. The goal of this section, is to investigate if the scalability and performance of applications which use collective I/O operations are improved when *LA_ TwoPhase I/O* is used instead of *Two_ Phase I/O*.

The fourh section reports the performance results for the techniques presented in Sections 4.2 and 4.3. As explained in 4.3, the *Adaptive-CoMPI* technique has different strategies in order to decide whether to apply compression or not: Runtime Adaptive Compression strategy (RAS) and Guided strategy (GS). In this section, first we perform a study of the RAS strategy and compare the results with original MPICH and also with the *CoMPI* technique by using different real applications and benchmarks. The goal is to show the advantages of transfering the data compressed only when it is worthwhile by using the best compression algorithm per message.

Second, we evaluate the GS strategy and compare the experimental results with RAS strategy. In this chapter, we aim to investigate if the *Adaptive-CoMPI* reduces the overall execution time and enhances the scability of applications.

Finally, the fifth section explains the results for the *Dynamic-CoMPI* technique detailed in Section 5.2. This technique integrates the *LA-TwoPhase I/O* and the *Adaptive-CoMPI* techniques in the same MPI implementation with the least possible overhead. The goal of *Dynamic-CoMPI* is to reduce at the same time, the volume and number of communications and the number in I/O, increasing the performance and scalabilty of parallel applications. Therefore, in this section we aim to evalutate if both techniques complement each other by using different applications, benchmarks and clusters.

As follows we sumarize the applications and benchmarks used in this thesis:

- Real-life applications:
  - *BIPS3D*: This is a 3-dimensional simulator of BJT and HBT bipolar devices [LGT03]. The application uses *Two_ Phase I/O* to write the results in disk. It is an irregular application.
  - *PSRG*: This is a parallel segmentation tool that processes grey-scale images [PSR06].
  - *STEM-II*: This is an Eulerian numerical model to simulate the behavior of pollutant factors in the air [MMD+04]. We have modified STEM-II in order to write the results by using the *Two_ phase I/O* technique. It is a regular application.
- Benchmarks:
  - The *NAS Parallel Benchmarks*: They are a set of benchmarks for performance evaluation of massively parallel computers [BHS+95].

We have performed our experiments on different clusters. A remarkable aspect is the fact that the employed clusters have three types of nodes: Single Core, Dual Core and Tetra Core. The main features of the clusters used for evaluations are:

- Cluster A: Magerit cluster, installed in the CESVIMA supercomputing center. Magerit has 1200eServer BladeCenter JS202400 nodes. Each node has two processors IBM 64 bits PowerPC *Single core* 970FX running at 2.2 GHz, and having 4 GB RAM and 40 GB HD. The interconnection network is Myrinet.
- Cluster B: A cluster with 22 *Dual Core* nodes, installed in the labs at University Carlos III. Each node has an Intel(R) Xeon(R) CPU with 4 GB of RAM. The network used is Gigabit.
- Cluster C: A cluster with 4 *Tetra Core* nodes, installed in the labs at University Carlos III. Each node has an Intel(R) Xeon(TM) CPU with 16 GB of RAM. The network used is Gigabit.

- Cluster D: A cluster with 64 *DualCore* nodes, installed in the labs at University Carlos III. Each node is an AMD with 512 MB of RAM. The network used is Ethernet.

Note that, the optimization proposed for *Two_ Phase I/O* in Chapter 3 cannot be applied to applications that do not use collective I/O operations. In contrast, the compression techniques presented in Chapter 4 can be applied at any kind of MPI application.

All techniques developed in this thesis are been implemented modifying by MPICHGM-1.2.7.15NOGM distribution. Therefore, we have a different implementation of MPICHGM-1.2.7.15NOGM for each technique:

- MPICHGM-1.2.7.15NOGM-LA_TwoPhase: In this implementation, the *LA_ TwoPhase I/O* technique is introduced by modifying the ADIO layer.
- MPICHGM-1.2.7.15NOGM-CoMPI: In this implementation, the *CoMPI* technique is introduced by modifying the ADI layer.
- MPICHGM-1.2.7.15NOGM-Adaptive-CoMPI: In this implementation, the *Adaptive-CoMPI* technique is introduced by modifying the ADI layer.
- MPICHGM-1.2.7.15NOGM-Dynamic-CoMPI: In this implementation, the *Dynamic-CoMPI* technique is introduced by modifyng the ADI and ADIO layers.

6.2 Real World Applications

6.2.1 BIPS3D Simulator

BIPS3D is a 3-dimensional simulator of BJT and HBT bipolar devices in [LGT03]. The goal of the 3D simulation is to relate electrical characteristics of the device with its physical and geometrical parameters. The basic equations to be solved are Poisson's equation and electron and hole continuity in a stationary state.

Finite element methods are applied in order to discretize the Poisson equation, hole and electron continuity equations by using tetrahedral elements, as shown in Figure 6.1. The result is an unstructured mesh. In this work, we have used four different meshes, as described later.

Using the METIS library [KK98], this mesh is divided into sub-domains, in such a manner that one sub-domain corresponds to one process, as shown in Figure 6.2. In this Figure, we can observe the mesh division in 7 sub-domains (one sub-domain per color). The next step is decoupling the Poisson equation from the hole and electron continuity equations. They are linearized by the Newton method. Then we construct for each sub-domain in a parallel manner, the part corresponding to the associated linear system. Each system is solved using domain decomposition methods. Finally, the results are written to a file.

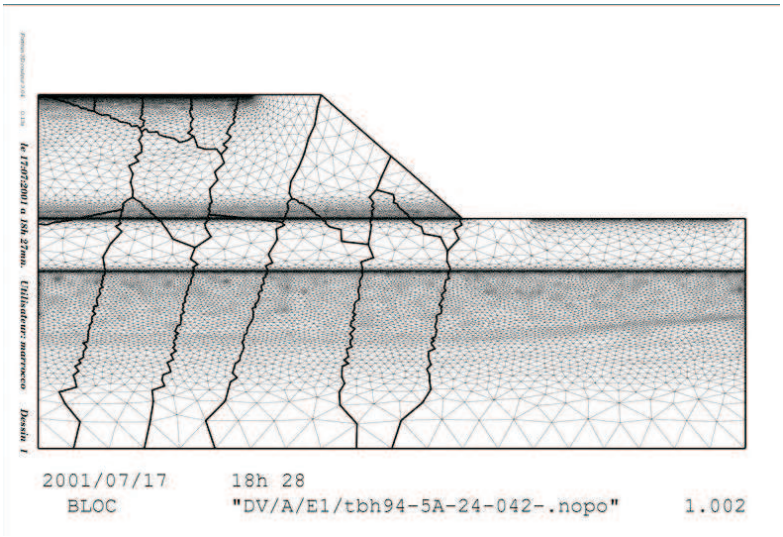


Figure 6.1: Discretization of a device.

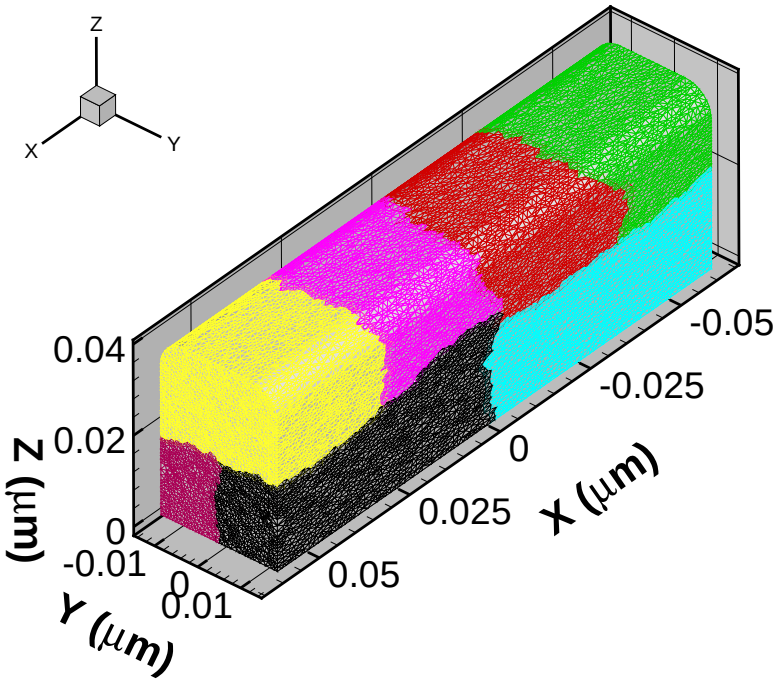


Figure 6.2: 3-dimensional simulation.

In the original BIPS3D version, the results are gathered at a root node, which stores the data sequentially to the file system. We have modified BIPS3D [FSI+07] to use collective writes during the I/O phase.

In the parallel I/O BIPS3D version, each compute node uses the distribution information initially obtained from METIS and constructs a view over the file. The view is based on an MPI data type. One example for a mesh distributed over two compute nodes is illustrated in Figure 6.3. Vector `v0` shows to which of the two processors the data is assigned. The first and second entries correspond to compute

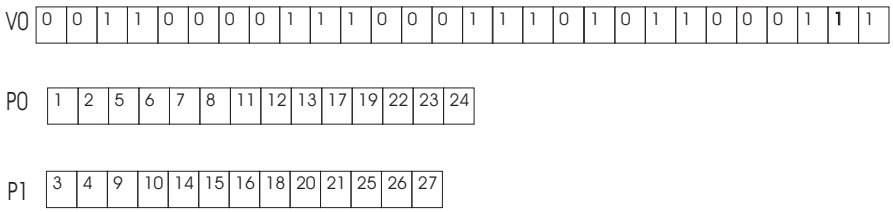


Figure 6.3: Example of data structures managed by BIPS3D.

node 0, the third to compute node 1, and so on. The vectors P0 and P1 contain the file positions where each of the elements of compute node 0 and 1 are to be stored. In order to achieve the MPI data type `MPI_Type_Indexed` is used. This data type represents non-contiguous chunks of data of equal sizes and with different displacements between consecutive elements.

Once the view on the common file is declared, the compute nodes write the data to its corresponding file part by using *Two_ Phase I/O* technique.

For our evaluation BIPS3D has been executed using four different meshes: *mesh*1 (47200 nodes), *mesh*2 (32888 nodes), *mesh*3 (732563 nodes) and *mesh*4 (289648 nodes). The BIPS3D associates a data stucture to each node of a mesh. The contents of these data structures are the data written to disk during the I/O phase. The number of elements that this structure has per each mesh entry is given by the *load* parameter. This means that, given a mesh and a load, the number of data written is the product of the number of mesh elements and the load. In this work we have evaluated different loads, concretely, 100, 200 and 500. Table 6.1 lists the different sizes (in MB) of each file based on load and mesh characteristics.

6.2.2 STEM

STEM-II [MSM+01] is an Eulerian numerical model to simulate the behavior of pollutant factors in the air. This model is being applied to the Power Plant of As Pontes (A Coruña, Spain) to study the relationships between the emissions, the atmospheric transport, the chemical transformations, the elimination processes, the resultant distribution of the pollutants in the air, and the deposition patterns. In addition, the STEM-II was chosen as a case study in the European CrossGrid project, proving its relevance for the scientific community for its industrial interest as well as its suitability for to high performance computing.

In terms of application performance, STEM-II is a computationally intensive

Mesh/Load	Mesh 1	Mesh 2	Mesh 3	Mesh 4
100	18	12	28	110
200	36	25	56	221
500	90	63	140	552

Table 6.1: Size in MB of each file based on the mesh and loads.

application that requires a multiprocessor environment for performing simulations in a reasonable response time.

There are two versions of STEM: Sequential and Parallel. We have choosen for our evaluations the parallel version explained in the work [SGC05], because, with this version, we not only can evaluate the performance of run-time data compression, but also the *LA_ TwoPhase I/O*, because it uses collective writes to disk.

6.2.3 PSRG

PSRG [PSR06] is a parallel segmentation tool that processes grey-scale images. This segmentation tool consists of two stages: a parallel seeded region growing algorithm (PSRG), and a region merging heuristic. In the first step, different segmentations, performed in parallel to the same input image are obtained. Each of these segmentations called partial segmentations, are also generated in parallel using a different number of processors. Next, a region merging heuristic is applied to the oversegmented image created as result of combining the different initial segmentations. The merging process is guided using only information about the behavior of each pixel in the initial segmentations (without external parameters).

The results of PSRG are written in disk by process 0. This means that the *LA_ TwoPhase I/O* technique not can be applied in this application because it does not use collective write. Thus, PSRG is used only to study the performance of *CoMPI* and *Adaptive-CoMPI* techniques.

6.3 NAS Benchmarks

The NAS parallel benchmarks (NPB) [BHS+95] were developed at the NASA Ames research center to evaluate the performance of parallel and distributed systems. The benchmarks, which are derived from computational fluid dynamics (CFD), consist of five parallel kernels (EP,MG, CG, FT, and IS) and three simulated applications (LU, BT, and SP). The simulated CFD applications use different implicit algorithms to solve 3-dimensional (3-D) compressible Navier-Stokes equations with minimun data traffic and computations in full CFD codes. The five kernels represent computational cores of numerical methods routinely used in CFD applications.

In our work, we use the MPI based implementation of these benchmarks, called NPB 2.3. A detailed description of the benchmarks can be found in [NAS].

A short description of the three simulated applications and five NAS parallel kernels follows:

- Simulated Application BT: This is a simulated CFD application which uses an Alternating Direction Implicit (ADI) approximate factorization to decouple the x, y, and z dimensions. The resulting system is Block Tridiagonal of 5x5 blocks which is solved sequentially along each dimension.

- Simulated Application SP: This is a simulated CFD application which employs the Beam-Warming approximate factorization. The resulting system of Scalar Pentadiagonal linear equations is solved sequentially along each dimension.

- Simulated Application LU: This is a simulated CFD application which uses the symmetric successive over-relaxation (SSOR) method to solve the discrete Navier-Stokes equations by splitting into blocks the Lower and Upper triangular systems.

- Kernel EP: This is an "embarrassingly parallel" kernel. It provides an estimate of the upper achievable limits for floating point performance, i.e., the performance without significant interprocessor communication. To do its job, this kernel generates pseudo-random floating point values according to a Gaussian and a uniform schemes.

- Kernel MG: This is simplified multigrid kernel. It requires highly structured long distance communication and tests both short and long distance data communication. It approximates a solution to the discrete Poisson problem.

- Kernel CG: This is conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix-vector multiplication.

- Kernel FT: This is a 3-D partial differential equation solution using FFTs. This kernel performs the essence of many "spectral" codes. It is a rigorous test of long-distance communication performance.

- Kernel IS: This is large integer sort. This kernel performs a sorting operation that is important in "particle method" codes. It tests both integer computation speed and communication performance.

To follow the evolution of computer performance, the NAS division had designed several classes of problems making kernels harder by modifying the size of data. For example, there are 6 classes of problems: S, W, A, B, C and D. Class S is the easiest problem and it is for test purposes only. Class D is the hardest one. Theses classes are available for all the kernels. For our evaluations, we have used class C.

6.4 Evaluation of LA_TwoPhase I/O

We have evaluated *LA_TwoPhase I/O* using BIPS3D and STEM-II applications, because in both of them the *Two_Phase I/O* is used to write data to disk. In the following subsections, we detail the evaluation of *LA_TwoPhase I/O* by using these applications. Note that, we define speedup as the ratio between the execution time by

using standard MPICH distribution and MPICHGM-1.2.7.15NOGM-LA_TwoPhase (see equation 6.1).

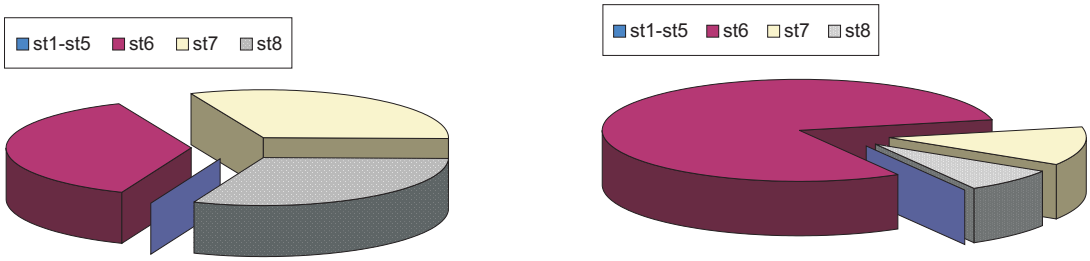
$$Speedup = \frac{Execution\_time\_MPICH}{Execution\_time\_LA\_TwoPhaseI/O} \quad (6.1)$$

6.4.1 *LA_ TwoPhase I/O* by using BIPS3D

Firstly, we evaluate *LA_ TwoPhase I/O* technique by using BIPS3D, in *Magerit* cluster (cluster A). We only use one proccessor per node. As follows, we summarize the *Two-Phase I/O* stages:

- *Offsets and lengths calculation (st1)*: In this stage the lists of offsets and lengths of the file are calculated.
- *Offsets and lengths communication (st2)*: Each process communicates its start and end offsets to the other processes. In this way, all processes have global information about the involved file interval.
- *File domain calculation (st3)*: The I/O workload is divided among processes. This is done by dividing the file into file domains (FDs). In this way, in the following stages, each aggregator collects and transfers to the file the data associated to its FD.
- *Dynamic Aggregator Pattern (st4)*: This stage only exists it in *LA_ TwoPhase I/O*. First, each process calculates the number of data of each FD that it has locally stored. After that, each aggregator is assignated to processes by applying *Linear Assignment Problem* (see Figure 3.8).
- *Access request calculation (st5)*: This calculates the access requests for the file domains of remote processes.
- *Metadata transfer (st6)*: This transfers the lists of offsets and lengths.
- *Buffer writing (st7)*: Data are sent to appropriate processes (see Figure 3.9).
- *File writing (st8)*: Data are written to file (see Figure 3.10).

The buffer and file writing stages (st7 and st8) are repeated as many times as the following calculus indicates: the size of the file portion of each process is divided by the size of the *Two-Phase I/O* buffer (4 MB in our experiments). First, the write size of each process is obtained by dividing the size of the file by the number of processes. For example, for *mesh*4 with load 500 and using 8 processes the size of the file is 552 MB (see Table 6.1). Therefore, the write size of each process is 69 MB. Then, the file size related to each process is divided into the buffer size of *Two-Phase I/O*. Consequently, the number of times is given by this value divided by 4 MB. For this example is 18.



(a) (b)

Figure 6.4: Stages of Two-Phase I/O for *mesh*1: (a) with load 100 and 16 processes and (b) with load 100 and 64 processes.

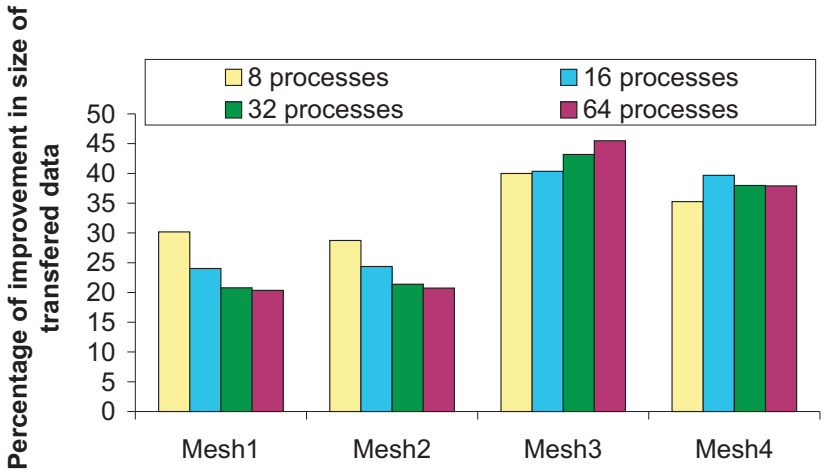
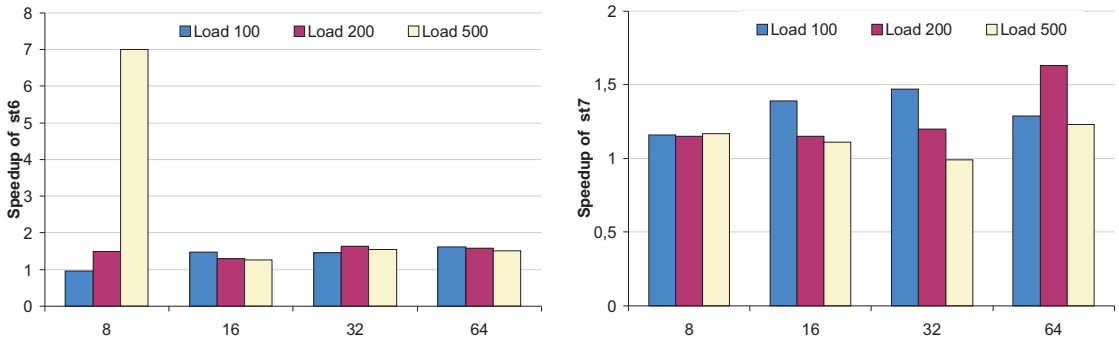


Figure 6.5: Percent reduction of transferred data volume for *mesh*1, *mesh*2, *mesh*3, and *mesh*4.

Figure 6.4 represents the percentage of time of *Two-Phase I/O* for *mesh*1 with load 100, with 16 and 64 processes, respectively. The costs of stages st1, st2, st3, st4 and st5 have been added up, and we have represented this value in the figures as st1-st5.

As we can see in Figures 6.4(a) and 6.4(b), the slowest stages are st6 and st7. Note that the cost of the st6 stage increases with the number of processes. These figures show the weight of the communication stages in the *Two-Phase I/O* technique. Moreover, we can see that the cost of these stages increases with the number of processors. Based on this, we conclude that st6 stage represents an important bottleneck in this technique.

With the appropriate aggregator pattern, the number of communications performed in stages st6 and st7 by using *LA_TwoPhase I/O* is reduced. Thus, this



(a) (b)

Figure 6.6: Speedup for *mesh*1: (a) in Stage 6 and (b) in Stage 7.

strategy reduces the number of communication operations because each aggregator increases the amount of locally assigned data. Figure 6.5 illustrates the percentage of reduction in communications for *LA_ TwoPhase I/O* over *Two-Phase I/O* for *mesh*1, *mesh*2, *mesh*3 and *mesh*4 and different numbers of processes. Therefore, we can see that when *LA_ TwoPhase I/O* is applied, the volume of transferred data is considerably reduced.

Once demonstrated that *LA_ TwoPhase I/O* reduces the number of communications performed in stages st6 and st7, we wish to evaluate if LA_TwoPhase I/O reduces the overall time of *Two-Phase I/O*. Therefore, we have measured the execution time of stages st6 and st7 for *mesh*1 with the orignal *Two_ Phase I/O* and with our technique. The speedup of these stages is calculated applying equation 6.1. Figure 6.6(a) shows the speedup of stage st6 for *mesh*1, for different loads, and a different number of processes.

In this figure we can see that the time of stage st6 is significantly reduced in most of cases. In this stage each process calculates what requests of other processes lie in its file domain and creates a list of offset and lengths for each process (which has data stored in its FD). In addition, in this stage, each process sends the offset and length lists to the rest of the processes. In *LA_ TwoPhase I/O*, much of the data that each process has stored belong to its FD (given that data locality is increased) and therefore fewer offsets and lengths are communicated.

Figure 6.6(b) illustrates the speedups of stage st7. Note that, again, this time is reduced in most cases. This is because in this stage, each process sends the data that has been calculated in st6 stage to the appropriate processes. In *LA_ TwoPhase I/O*, much of the data that each process has stored belong to its FD, therefore, it send less data to the other processes, reducing the number of transfers and the data volume.

Figure 6.7 shows the overall speedup of our technique for *mesh*1, *mesh*2, *mesh*3 and *mesh*4 with 8, 16, 32 and 64 processes. Each speedup shown in Figure 6.7 is calculated by comparing the original one with our technique. Then, equation 6.1 is applied by using the values calculated before.

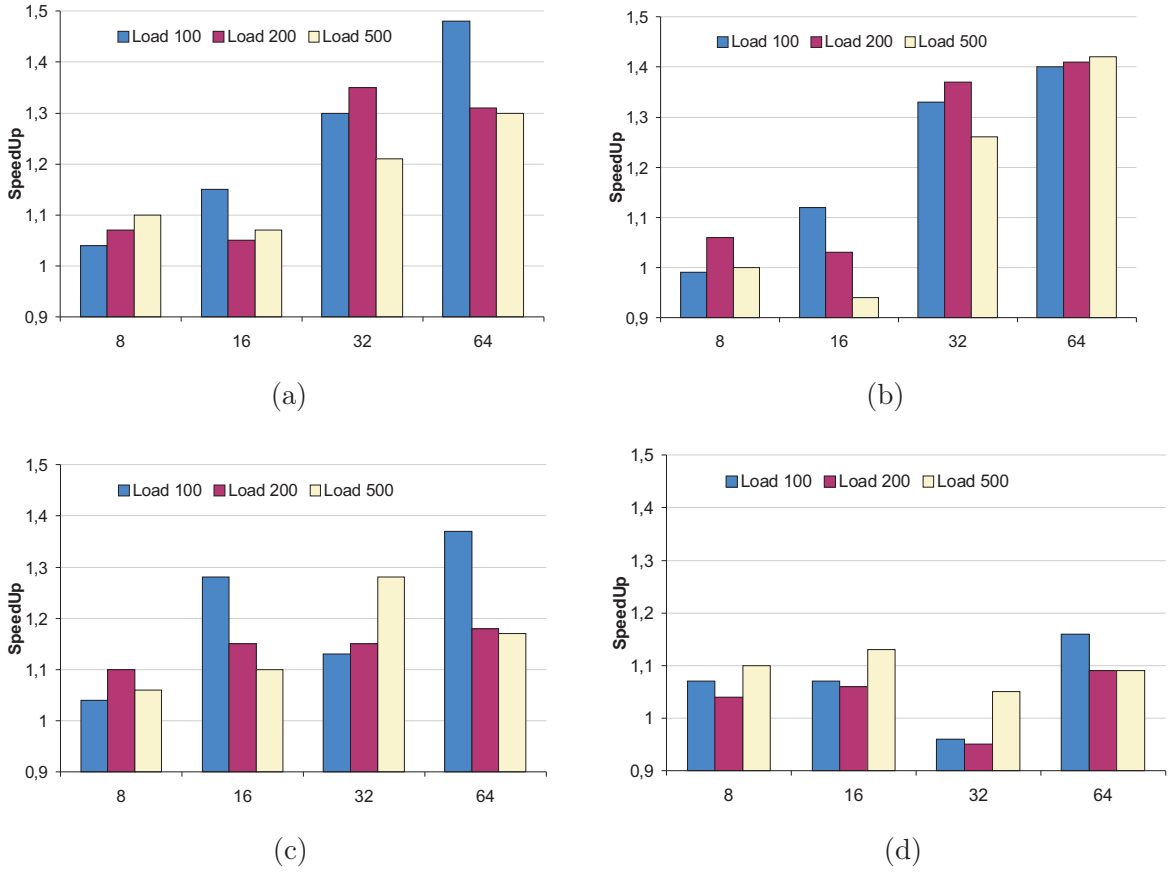


Figure 6.7: Overall speedup for: (a) *mesh*1 (b) *mesh*2 (c) *mesh*3 and (d) *mesh*4.

BIPS3D has an irregular data distribution that changes by using a different number of processes. So, the processes have to perform many communications in *Two_Phase I/O* to write contiguous regions of data into the file. Figure 6.7 shows a significant improvement in the execution time for the *LA_TwoPhase I/O* technique in most cases.

In this figure, we have included overall execution time for obtaining the speedups. For this reason, the speedup is smaller than in previous stages. Nevertheless, we can notice that in most cases there is a significant improvement in the execution time for *LA_TwoPhase I/O* technique. The original technique performs better in 4 of the 48 cases, but the loss was near 1 in all of them. It appears that, for these cases (which represent less than 10% of the total), the data locality seems to be good enough in the original distribution and the additional cost to find a better distribution did not pay off.

It is important to emphasize that the additional cost of the new stage (st4) is very small when compared with the total time. The fraction of this stage in the overall execution time is small: in the best case it is 0.07% of the time (*mesh*2, 8 processes, and load 500) and in the worst case it is 7% (*mesh*3, 64 processes, and load 100).

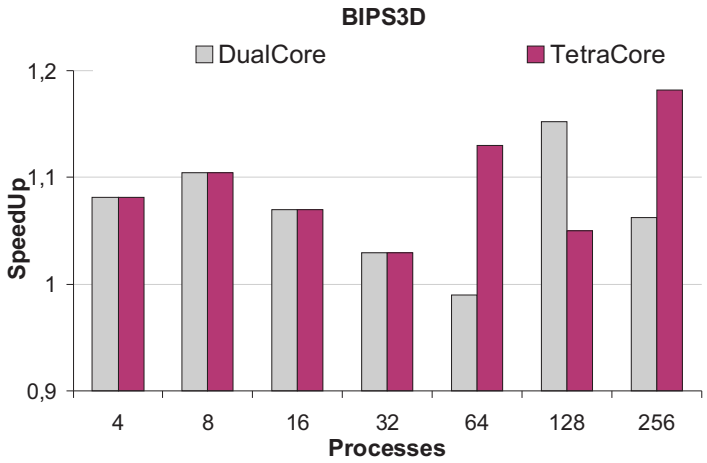


Figure 6.8: Execution time improvement of BIPS3D by using mesh 4 and load 100: Dual Core and Tetra Core processors.

We have also evaluated the BIPS3D application by using *LA_ TwoPhase I/O* with the *clusters B* and *C*. As follows, we illustrate in Figure 6.8 the speedups obtained in each cluster, for *mesh*4 and load 100.

If we compare the speedup obtained for *mesh*4 with load 100 by using the three clusters (see figures 6.7 and 6.8), we observe that:

- With Gigabit network, the improvement is higher than with Myrinet. This is because Myrinet is faster than Gigabit, so the impact of communications is minor in this network. Nevertheless, with the three architectures, we obtain quite good results applying *LA_ TwoPhase I/O*.
- The cluster architecture also has a significant impact in the achieved speedup. The number of aggregators when Tetra Core architecture is used, is just half that with Dual Core architecture, because *Two_ Phase I/O* chooses by default a single core per node as aggregator. Therefore, with Tetra Core architecture, choosing a good aggregator pattern has a great impact on the performance. Thus, the speedup of *LA_ TwoPhase I/O* in some cases is highest when Tetra Core architecture is used, as we can see in Figure 6.8 (64 and 256 processes).

6.4.2 *LA_ TwoPhase I/O* by using STEM-II

To evaluate *LA_ TwoPhase I/O* with STEM application, we have used *clusters B* and *C* with a different number of processes, as shown in Figure 6.9. STEM-II is a regular application. So, in most cases, the aggregator pattern found by *LA_ TwoPhase I/O* by using Dual Core cluster is the same as the original pattern. For these reasons, the speedup achieved by using *LA_ TwoPhase I/O* with 8, 16 and 32 is one, as illustrated in Figure 6.9. On the other hand, when Tetra Core architecture is used with STEM-II application, the *LA_ TwoPhase I/O* technique finds a better aggregator pattern in most cases. So, once again, we demonstrate that the cluster

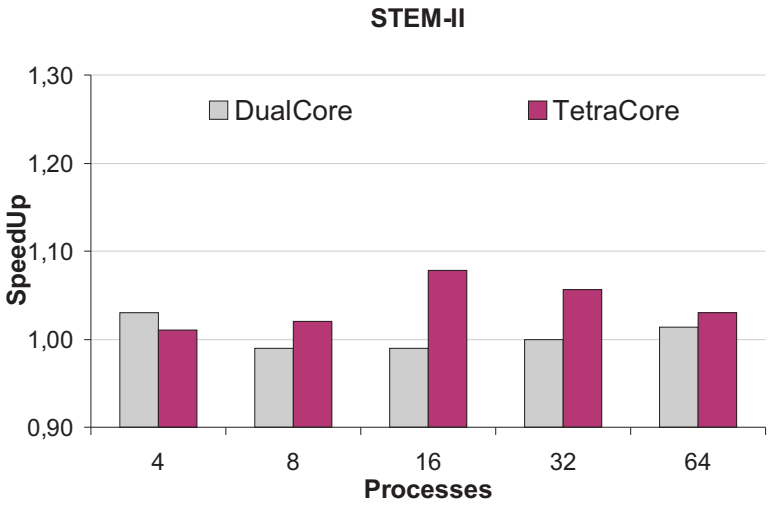


Figure 6.9: Execution time improvement of STEM-II by using Dual Core and Tetra Core processors.

architecture has a great impact on the achieved speedup. As explained before, the number of aggregators when Tetra Core architecture is used, is just half that with Dual Core architecture. Thus, the speedup of *LA_ TwoPhase I/O* in most cases is higher when Tetra Core architecture is used, as we can see in Figure 6.9.

6.5 Adaptive-CoMPI Evaluation

This section is dedicated to study the performance of compression strategies described in Chapter 4. The main objetive of this evaluation is to compare the speedups of all applications described in Sections 6.2 and 6.3 for both the original and the MPICHGM-1.2.7.15NOGM-Adaptive-CoMPI with a different number of processes, different compression algorithms, and different strategies.

For a detailed study of the *Adaptive-CoMPI* technique, we have chosen the *cluster D*, which is composed of 64 DualCore nodes. The network used is an Ethernet. Note that, for this evaluation, we only want to use one core per node.

Furthemore, we aim to show the improvement of activating and deactivating compression over all messages. Therefore, we compare the speedups of *Adaptive-CoMPI* (by using RAS strategy) over *CoMPI*. Note that *CoMPI* always sends the data compressed. In addition, *CoMPI* uses the same compressor to compress all messages transferred, which have been previously selected by a MPI_Hint. Moreover, *Adaptive-CoMPI*, can fit any particular application, because its implementation is transparent for users, and it integrates different compression algorithms. Furthermore, compression is turned on and off and the most appropriate compression algorithms are selected in run-time depending on the characteristics of each message.

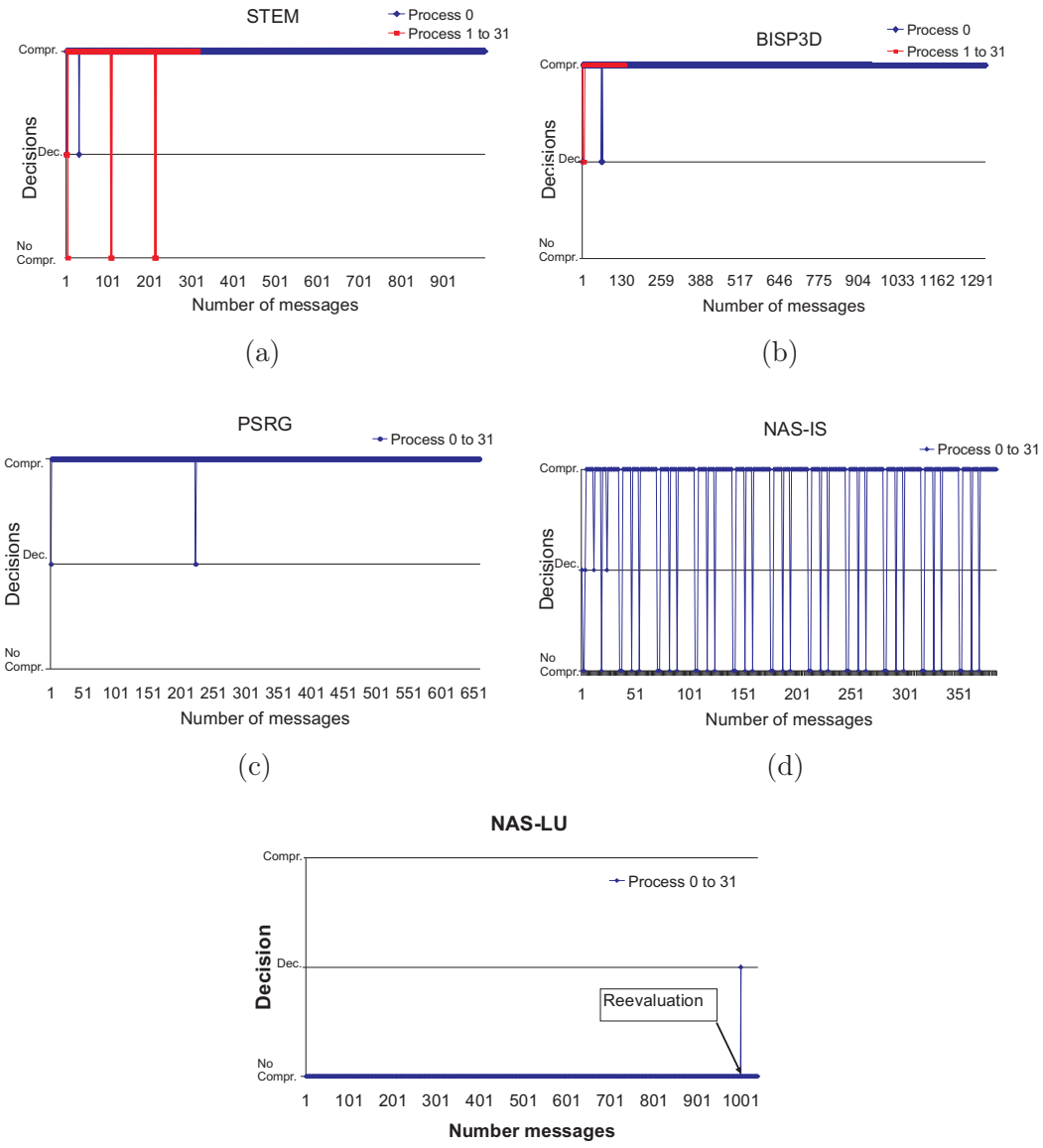


Figure 6.10: Compression decision with 32 processes: No compress, deciding (labeled as Dec), Compress.: (a) STEM (b) BIPS3D (c) PSRG (d) NAS-IS (e) NAS-LU.

6.5.1 Runtime Adaptive Compression Strategy Evaluation

As we explained in Section 4.6, RAS allows enabling and disabling dynamically the compression in order to reduce the applications' execution time. Figure 6.10 shows the different decisions that the runtime strategy takes for each application at execution time. These decisions can be No compress (*No_ compr.*) which means that the process sends the message without compression, deciding (*Dec.*) which means that the process calculates the message speedup to update the value of one of the two compression boundaries (*length_ yes_ compression* or *length_ no_ compression*), and

Compress (*Compr.*) which means that the process sends the message compressed.

For each iteration of BIPS3D and STEM-II applications, the process zero sends data to the other processes. Therefore, the number of messages that are sent by process zero in these applications is higher than for the other processes.

As shown in Figure 6.10, the first decision for any process is to find the size of the message from which it obtains a benefit by compressing data, storing the message size as one of the two datatype limits: *length_yes_compression* or *length_no_compression*. For subsequent messages with the same datatype, the possible decisions are:

- To send data compressed if the message size is larger than *length_yes_compression*.
- To send data uncompressed if the message size is smaller than *length_no_compression*.
- To update one of the two limits if the message size is between the two marks.

This procedure is applied to every unclassified message. As we can see in Figure 6.10, RAS adapts to the application behavior. For instance, for STEM, BIPS3D and PSRG applications, there are no changes in the decision because the application behavior is uniform. In contrast, for NAS-IS, RAS decides to compress only a portion of the messages.

In addition, another characteristic of the RAS is that a process can reevaluate the datatype study for two reasons. The first reason is that the number of messages (with the same datatype) sent without compression, as shown in Figure 6.10(e) for NAS-LU application, is higher than that defined by the reevaluation threshold. The reason for this reevaluation is to check if the process has set a very high value of *length_no_compression* (value higher than the message sizes), making RAS always choose no compression. The second reason to reevaluate is to have several decision errors in a short time interval, as shown in Figure 6.11 for STEM application. Process 30, taking advantage of the reevaluation, soon changes its initial decision of compressing with RLE algorithm, starting to send data compressed with LZO algorithm.

The average compression ratio for PSRG is 80%, for STEM it is 30%, for BIPS3D it is 40%, for NAS-IS it is between 5%-18% and NAS-LU it is 5%. The different compression ratios of the applications depend on the type of data and redundancy levels. For instance, RAS has detected that the cost of compressing the NAS-LU messages is greater than the advantage of sending a small amount of data. Therefore, all NAS-LU messages are sent without compression. For NAS-IS, RAS has decided that it is better to send the message compressed when the compression ratio is higher than 15%. For BIPS3D, PSRG and STEM applications almost all the messages are sent compressed.

Figure 6.12 shows the execution time improvement for all the considered benchmarks by using adaptive compression (*Adaptive-CoMPI*), and without deactivating compression (*CoMPI*). Speedup for Figure 6.12(a) is defined (Equation 6.2) as the

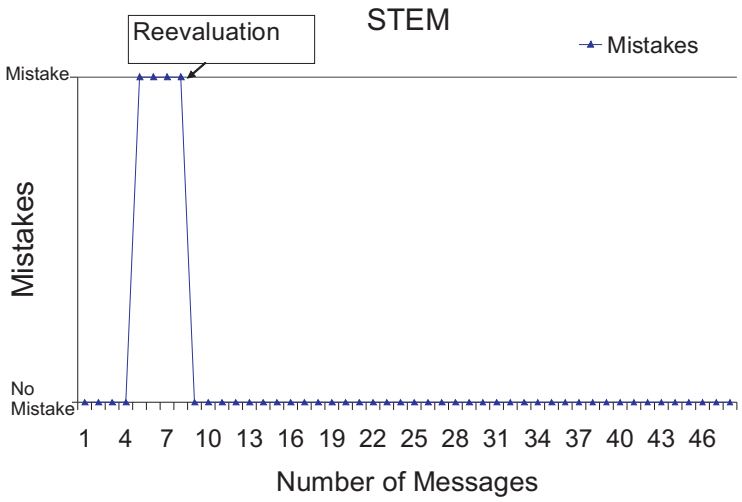


Figure 6.11: Reevaluation with STEM.

ratio between the overall execution time using the standard MPICH distribution and the *Adaptive-CoMPI* implementation (based on the RAS Strategy). On the other hand, the speedup for Figure 6.12(b) is defined (Equation 6.3) as the ratio between the overall execution time using the standard MPICH distribution and CoMPI distribution. Values greater than one imply a reduction in the execution time using compression. Note that we consider the overall execution time (computation, communication, and I/O) of the benchmarks. It is important to note that all the execution time improvement is related to the reduction of the communication time.

$$Speedup = \frac{Overall\_execution\_time\_MPICH}{Overall\_execution\_time\_Adaptive - CoMPI} \tag{6.2}$$

$$Speedup = \frac{Overall\_execution\_time\_MPICH}{Overall\_execution\_time\_CoMPI} \tag{6.3}$$

In our previous approach (*CoMPI*), it was mandatory to specify (by using MPI hints) the algorithm used for the compression, due to the fact that in this implementation it is not possible to find the best compressor per message. In addition, in the CoMPI distribution, the compression is never deactivated. We have chosen the LZO compressor for CoMPI in these experiment, because in most of the cases it compresses more efficiently than the other compression algorithms. We can observe that, in all cases, the performance of *Adaptive-CoMPI* is higher (or equal to) than *CoMPI*. In addition, the *Adaptive-CoMPI* speedups are never less than 1. These results are due to two main reasons:

- Adaptive-CoMPI is able to deactivate the compression when it is not worthwhile. This is clearly reflected in the results for NAS benchmarks.

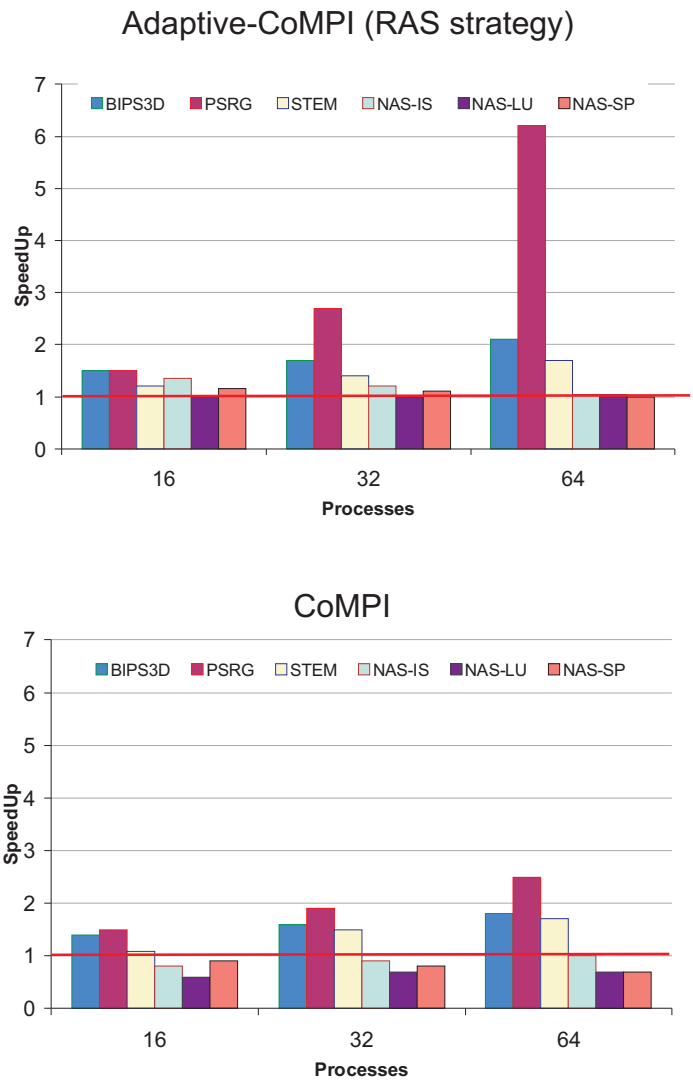


Figure 6.12: Execution time improvement of applications BIPS3D, PSRG, STEM, NAS-IS, NAS-LU and NAS-SP, by using: (a) Adaptive-CoMPI (b) CoMPI.

- Adaptive-CoMPI chooses the best algorithm per message. For example, in the case of PSRG application, Adaptive-CoMPI chooses for some messages the RLE algorithm and for others the LZO. This selection depends on the specific characteristics of each message.

When considering the performance of IS benchmark executed with 16 processes, we can observe that, in the case of CoMPI distribution, the speedup obtained is less than 1, while Adaptive-CoMPI gets a speedup of 1.3. This behaviour is due to the fact that Adaptive-CoMPI has detected that only 42% of messages exchanged are worth compressing. The remaining messages are sent uncompressed. This reduces the overhead that CoMPI generates sending all messages compressed.

We have evaluated all NAS benchmarks using Adaptive-CoMPI, and in all cases (except IS and SP) the compression is disabled by *Adaptive-CoMPI*. In decision, it

is given that the transferred messages in most of the NAS benchmarks (LU, CG, SP, MG, BT and EP) are of IEEE 754 double precision type. This kind of messages is very difficult to compress by all the compression algorithms that can be found in the literature.

In Table 6.2 we can see the compression ratio, the compression and decompression time, and the speedup (calculated as shown in Equation 4.6). Results are displayed for all the algorithms included in the Adaptive-CoMPI Library, and they refer to a 300 KB message of IEEE 754 double precision type extracted from the CG benchmark.

Table 6.2 shows that none of the compressors used reaches a speedup greater than 1. FPC and Huffman algorithms compress 6% of the message, but the time required to compress and decompress is too high. For these reasons, *Adaptive-CoMPI* disables the compression in LU, BT, CG, MG, and EP benchmarks.

We emphasize that the higher the number of processes, the greater the application speedup achived by *Adaptive-CoMPI*. This behavior is due to the increasing number of communications. Therefore, the improvement of the communication performance produces a greater impact on the overall application performance. Thus, we can conclude that the scalability is also enhanced with Runtime Adaptive Compression.

6.5.2 Guided Strategy Evaluation

When the Guided strategy has to build the *Decision-files*, it compresses and decompresses each message stored in the *Trace-files* with all algorithms included in *Compression-Library*. Then, it estimates the time to send the compressed and uncompressed messages. Finally, the strategy calculates the speedup per message by using Equation 6.4. However, in order to calculate the speedup, RAS has to estimate the time to send the messages compressed and uncompressed, and also the time to decompress messages. Moreover, the Guided strategy takes each decision (compress or not, and the employed compression algorithm) studying each message separately. RAS instead learns at execution time and it takes the same decision for all messages with the same characteristics.

$$Speedup = \frac{Time\_to\_send\_message}{Time\_to\_send\_message\_compressed + T\_compress + T\_decompress} \tag{6.4}$$

Algorithm	Compression_Ratio(%)	Compression_Time(μs)	Decompression_Time (μs)	SpeedUp
FPC	6.3	24717.8	23874.6	0.37
LZO	0	8191.0	414.9	0.76
RLE	0	2537.4	1146.1	0.88
HUFF	6.05	13033.9	21292,6	0.45
RICE	0	59993.3	8146,3	0.28

Table 6.2: Compression study for IEEE 754 double precision type by using a 300 KB message transferred in CG Benchmark.

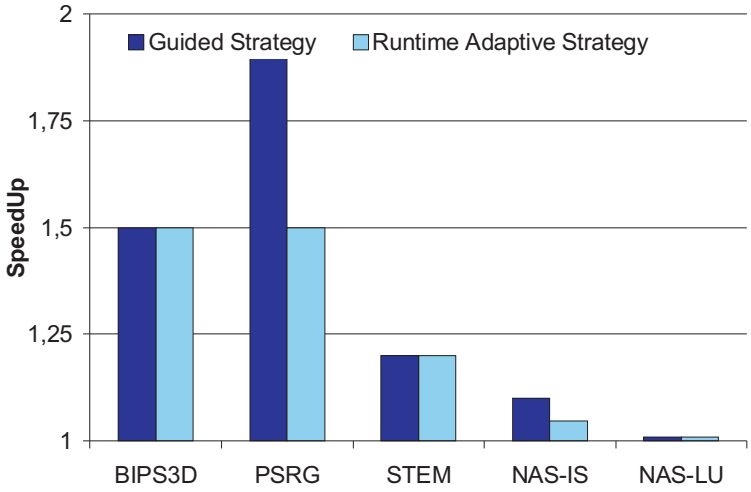


Figure 6.13: Speedup with Guided and Runtime strategies by using 16 processes.

Figure 6.13 shows the speedup achieved using *Adaptive-CoMPI* with GS and RAS strategies when BIPS3D is executed with 16 processes. For each test, we first have measured the speedup by using RAS. Next, we have executed each test with an MPI hint called "GS hint" to generate *Decision files*. Finally, we have measured the speedup by using the Guided strategy applying the decisions previously calculated. Note that this approach uses the optimal compression algorithm for each message (including non-compression of data).

The speedup is defined as the ratio between the original *MPICH* and the *Adaptive-CoMPI* execution times. A value higher than one indicates that compression reduces the application execution time. In contrast, a value smaller than one implies an increment of the execution time when compression is employed. It can be seen in Figure 6.13, some applications such as PSRG and NAS-IS, show an increment in the speedup when the Guided strategy is used. In general, the speedups achieved with GS and RAS strategies are very similar. This means that our RAS strategy is almost as good as the GS strategy, and that the decisions of the adaptive strategy are very accurate. However, GS could be beneficial in applications showing very irregular message patterns, where RAS could expend more time reevaluating decisions.

6.6 Dynamic-CoMPI Evaluation

As we explained in Chapter 5, *Dynamic-CoMPI* is designed as an extended MPICH. *Dynamic-CoMPI* uses data compression to reduce the volume of communications. It also searches for a new aggregation pattern to reduce the number of communications performed during the collective I/O. Thus, we have integrated *LA-Two-Phase I/O* and *Adaptive-CoMPI* techniques in the same implementation of MPICH to develop *Dynamic-CoMPI*.

In Sections 6.4 and 6.5 we have analyzed *LA-Two-Phase I/O* and *Adaptive-CoMPI* techniques separately. This section is dedicated to evaluate the performance of *Dynamic-CoMPI*, that has integrated both techniques. So, for a detailed study, we compare the speedup of the three techniques by using the same conditions: applications, number of processes and platform characteristics.

Nowadays, many cluster architectures are based on multi-core CPU, which means placing two or more processing cores on the same chip. Multi-core architectures allows faster execution of applications by taking advantage of parallelism.

Optimization techniques introduce some overhead because of the extra task that has to be done. But in multi-core systems some of these tasks can be done in parallel, so the overhead can be dramatically reduce. This motivation guided our research, and we aim to study the behavior of our techniques in two multi-core clusters by using Dual Core and Tetra Core processors. Therefore, we have choosen for *Dynamic-CoMPI* evaluation clusters B and C, which are the ones with these characteristics. The network used is a Gigabit for both clusters.

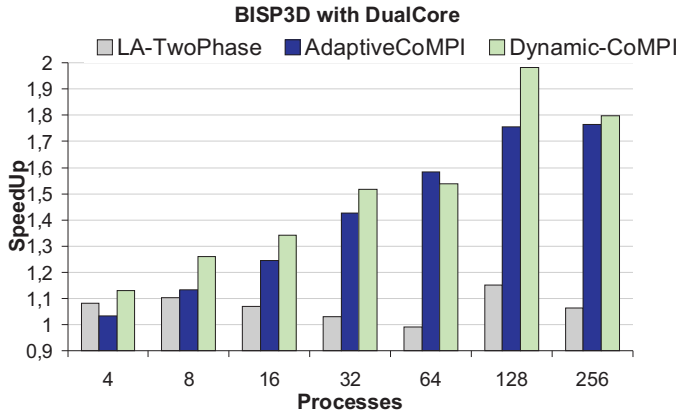
6.6.1 Dynamic-CoMPI by using Real World Applications

The main goal of this evaluation is to evaluate if *LA_ TwoPhase I/O* and *Adaptive-CoMPI* techniques can be applied together without overhead. For these reason, we have chosen STEM and BIPS3D real world applications for those evaluations, because both applications perform communications among the processes, and they write their results into file by using *Two_ Phase I/O*.

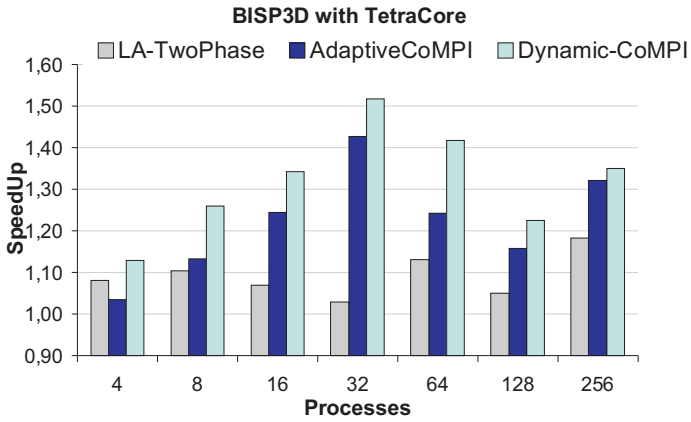
Figure 6.14 shows the different speedups achieved by BIPS3D when *LA-Two-Phase I/O* and *Adaptive-CoMPI* techniques are applied separately, and when they are applied together (*Dynamic-CoMPI*) by using different clusters and number of processes. The speedup for *Dynamic-CoMPI* technique is defined as the ratio between the execution time of the original *MPICH* and the MPICH-1.2.7NOGM-Dynamic-CoMPI (see equation6.5). A value higher than one indicates that our technique reduces the application execution time. In contrast, a value smaller than one implies an increment of the execution time when our technique is used. The speedup for *LA_ TwoPhase I/O* and *Adaptive-CoMPI* is defined in equations 6.1 and 6.2 respectively.

$$Speedup = \frac{Overall\_execution\_time\_MPICH}{Overall\_execution\_time\_Dynamic-CoMPI} \tag{6.5}$$

First we have evaluated the *LA-Two-Phase I/O* technique by using BIPS3D with clusters B and C, and with a different number of processes. The speedup of *LA-Two-Phase I/O* depends on the data locality. This means that if the default aggregators pattern is good enough, the additional cost to find a better distribution is not worthwhile, as illustrated in Figure 6.14(a) with 64 processes. In this case, the speedup is one because the original pattern is the same found by *LA-Two-Phase I/O*.



(a)



(b)

Figure 6.14: Execution time improvement of BIPS3D by using: (a) Dual Core (b) Tetra Core processors.

In most cases, *LA-Two-Phase I/O* finds an aggregator pattern for BIPS3D better than the original, thus reducing the number of communications and therefore reducing the execution time.

Once evaluated *LA-Two-Phase I/O*, we have executed BIPS3D with *Adaptive-CoMPI*. The speedup of *Adaptive-CoMPI* depends on the compression ratio and specific characteristics of the application, such as the communication pattern, because if two processes are located on the same node, the compression is disabled. The different compression ratios of the applications depend on the datatype and redundancy level. For example, the average compression ratio for BIPS3D is 40%. So, for BIPS3D almost all the messages are sent compressed. For this reason, BIPS3D reduces the execution time by using *Adaptive-CoMPI*. In addition, the cluster architecture affects the results. As can be seen in Figure 6.14(a), the speedup of *Adaptive-CoMPI* in Dual Core nodes is higher than Tetra Core (Figure 6.14(b)). When the Tetra Core architecture is used, the number of processes in the same node increases,

and therefore the number of communications between different nodes is lower than in Dual Core architecture. This means that the compression in Tetra Core architecture is applied less often than in Dual Core architecture.

Figures 6.14(a) and (b) show that the speedup achieved with *Dynamic-CoMPI* is the speedup of *LA-Two-Phase I/O* plus the speedup of *Adaptive-CoMPI* in both clusters. That is, the performance gains of both techniques are accumulated, which means that both approaches are complementary.

We have also evaluated STEM application by using the three techniques with clusters B and C, and with different number of processes, as shown in Figures 6.15(a) and (b). As we explained, STEM is a regular application, so in most cases, the aggregators pattern found by *LA-Two-Phase I/O* by using Dual Core cluster is the same as the original pattern, as illustrated in Figure 6.15(a) with 8, 16 and 32 processes. On the other hand, when Tetra Core architecture is used with STEM application, the *LA-Two-Phase I/O* technique finds a better aggregators pattern in most cases, as we can see in Figure 6.15(b).

It is important to note that the average compression ratio for STEM is 30%. For this reason, the speedup of STEM using *Adaptive-CoMPI* in both clusters is less than BIPS3D using the same technique. This is because STEM does not send all messages compressed, and the volume of compressed messages is higher than BIPS3D. As shown in Figure 6.15(a), when the number of processes is 4, the *Adaptive-CoMPI* technique decides to not compress the messages, because the cost of sending messages with compression is higher than the cost to sending them without compression. For this reason, the speedup is equal to 1 in this case. Similar behaviour is shown in Figure 6.15(b) by using 4, 8, 16 and 64 processes.

Figures 6.15(a) and (b) also show, that the speedup of *Dynamic-CoMPI* with STEM is the speedup of *LA-Two-Phase I/O* plus the speedup of *Adaptive-CoMPI*. In all cases, the speedup of *Dynamic-CoMPI* is higher than one.

We can conclude that the *LA-Two-Phase I/O* and *Adaptive-CoMPI* techniques can be applied together with a minimal overhead, because *Dynamic-CoMPI* increases the speedup achieved by applying both techniques separately.

Once the *Dynamic-CoMPI* with STEM and BIPS3D applications analyzed, we study the performance of *Dynamic-CoMPI* when only the run-time compression is applied. The PSRG application was chosen for this study. In this application, the image is divided among the processes, and each process applies the segmentation operation on the portion of image that it has assigned. After that, the processes gather all the portions segmented in a single image. Finally, the process with rank 0 writes the segmented image. Therefore, in the PSRG application, we only can apply run-time compression, given that collective I/O is not used.

We have evaluated the speedup by comparing the PSRG execution time achieved with the original MPICH with *Dynamic-CoMPI* by using Dual Core and Tetra Core clusters (Figure 6.16).

Once again, the performance of our technique depends on the compression ratio, the size of message, and the communications pattern. The compression ratio

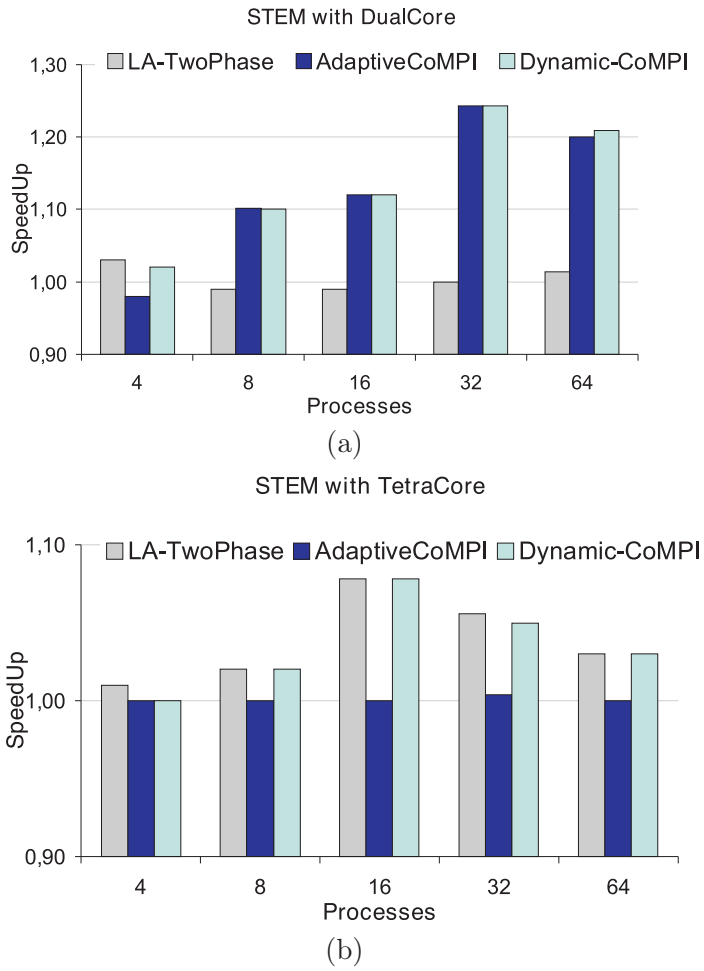


Figure 6.15: Execution time improvement of STEM by using: (a) Dual Core (b) Tetra Core processors.

for PSRG is 40%. Thus, almost all messages are sent compressed. When Tetra Core architecture is used, the number of processes in the same node is higher, and therefore the number of communications between different nodes is lower than in the Dual Core architecture. Therefore, PSRG application achieves better results by using *Dynamic-CoMPI* with Dual Core architecture, as shown in Figure 6.16. However,*Dynamic-CoMPI* achieves a speedup less than one in both clusters.

We emphasize that, the higher number of processes, the greater the application speedup in most of the cases. This behavior is due to the increasing number of communications. Therefore, the improvement of the communication performance produces a greater impact on the overall application performance. Based on that, we can conclude that our approach increases the application scalability.

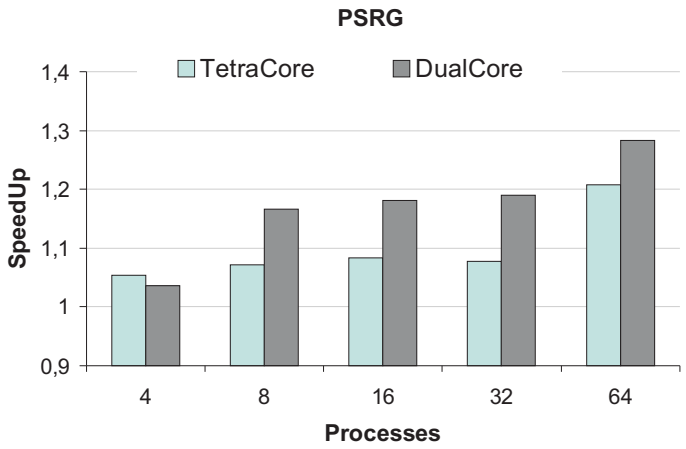


Figure 6.16: Speedup of PSRG by using Dual Core and Tetra Core processors.

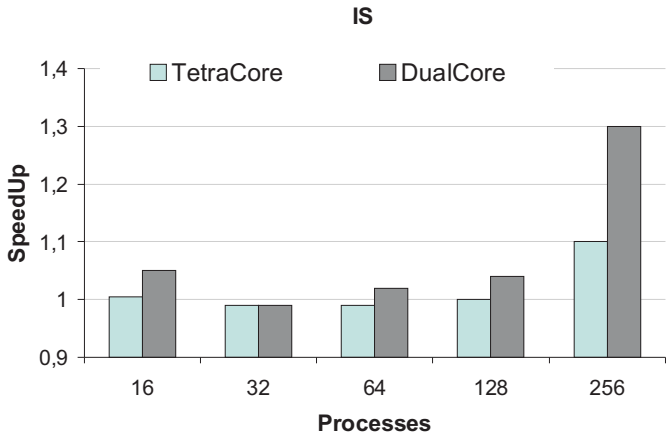


Figure 6.17: Speedup of IS by using Dual Core and Tetra Core processors.

6.6.2 Dynamic-CoMPI Evaluation by using NAS benchmarks

We have evaluated all the NAS benchmarks using the *Dynamic-CoMPI* technique, and in all cases (except IS) the compression is disabled. This decision is taken because data of the transferred messages in most of the NAS benchmarks (LU, CG, SP, MG, BT, EP) are of the IEEE 754 type. These kinds of messages are very difficult to compress by all the compression algorithms. Therefore, we show the evaluation of *Dynamic-CoMPI* technique by using the benchmark IS, where the data transferred are integers. The speedup of this benchmark is evaluated by comparing the execution time achieved with the original MPICH implementation and with *Dynamic-CoMPI* implementation. We have evaluated *Dynamic-CoMPI* by using both architectures and different number of processes.

Performance results are shown in Figure 6.17. In this figure we can observe that the speedup achieved by using *Dynamic-CoMPI* with both architectures is, in some cases, one. This behaviour is due to the fact that the average compression ratio for

IS messages is between 5% and 18%. The different compression ratios, depends on the redundancy level of messages. In this case, *Dynamic-CoMPI* decided to send the messages compressed only when the compression ratio is higher than 15%, because it detects that the cost of compressing when the compression ratio is less than 15%, and is greater than the advantage of sending the data uncompressed.

Note that when we evaluate IS using 256 processes the fastest speedup is achieved in both architectures (Figure 6.17). The reason is that with 256 processes, the number of messages transferred is very high, and they cause network saturation. Thus, the improvement of the messages compression produces a greater impact on the overall benchmark performance by enhancing system scalability.

Also we can see in Figure 6.17, that the best results are achieved in the Dual Core cluster. As explained before, with Tetra Core nodes, the compression is disabled more often. If two processes involved in a communication are in the same node, the message is sent without compression. For these reasons, *Dynamic-CoMPI* produce a better improvement on the Dual Core cluster.

Finally, we want to remark that for all evaluations performed (real-life applications and IS benchmark), the techniques presented in this paper achieve a speedup higher or very close to 1. This means, that our approach never losses performance when we applied our techniques.

6.7 Summary

In this chapter, we have evaluated all strategies proposed in this Ph.D. thesis by using different applications, benchmarks, network and cluster architectures:

- The *LA_ TwoPhase I/O* technique reduces the overall execution time of applications, because in most cases, *LA_ TwoPhase I/O* finds a better aggregator pattern than the original, reducing the number of communication performed in redistributed phase. We have demostrated that the cluster architecture also has a great impact in the achieved speedup. This impact in performance is greater when Tetra Core architecture is used because the number of aggregator is just half that with Dual Core architecture.

- *Adaptive-CoMPI* technique improves the speedups for many applications because the volume of communications is reduced by using the best compression algorithm per message. It also demonstrates that, even when compression is deactivated, application performance is never reduced due to the overhead of our solution. The results demonstrated that the *Runtime Adaptive Strategy* performance is as good as the *Guided Strategy* (off-line strategy). In addition, the runtime performance gain is bigger when more processes are employed, which increases scalability.

- The evaluation of *Dynamic-evaluation* technique shows that for all of the considered scenarios, the speedup of *Dynamic-CoMPI* is the speedup of *LA-Two-*

Phase I/O plus the speedup of *Adaptive-CoMPI*. That means that both techniques are complementary. We emphasize that, the higher the number of processes the greater the application speedup achieved *Dynamic-CoMPI* in most the cases. This behavior is due to the increasing number of communications. Therefore, the improvement of the communication performance produces a greater impact on the overall application performance.

Chapter 7

Main Conclusions

7.1 Main Conclusions

In this thesis we have proposed an optimization of the MPI communication library, which uses two strategies in order to reduce the impact of communications and I/O requests in parallel MPI-based applications. Those strategies are independent of the application and transparent to users. Both of them have been included in *MPICH* [GL97] implementation, developed jointly by Argonne National Laboratory and Mississippi State University. We have selected as a collective I/O technique to improve the *Two-Phase I/O* [RC95], extended by Thakur and Choudhary in *ROMIO* [ROM].

The main conclusions of this thesis are:

- **Reduction in the number of communications in collective I/O operations:** The first strategy is an optimization of the Two-Phase collective I/O technique from ROMIO. In order to increase the locality of the file accesses, we have proposed using the Linear Assignment Problem (LAP) to find an optimal I/O aggregator pattern. We have developed an optimization of *Two-Phase I/O* called *LA_ TwoPhase I/O*, where we have implemented the **new dynamic I/O aggregator pattern proposed**. We have shown that the *LA_ TwoPhase I/O* improves the overall performance, when compared to the original *TwoPhase I/O*.

  The new stage (st4), which has been added to the *LA_ TwoPhase I/O* technique has an insignificant overhead in comparison to the overall execution time. In the evaluation shown in Section 6.4 we have observed that the higher the number of processes, the greater the improvement provided by this technique. In addition, it is important to emphasize that *LA_ TwoPhase I/O* can be applied to every kind of data distribution. Also, we have evaluated

LA_ TwoPhase I/O by using different applications, clusters, and networks. Therefore, we can conclude that, in most cases, *LA_ TwoPhase I/O* is beneficial for a large field of application domains and different parallel architectures.

- **Reduction of transferred data volume**: The second strategy is based on applying run-time compression to the MPI. The novelty of this stratey is that we do not focus on developing new compression algorithms, but on selecting in runtime the most appropriated one in order to maximize the speedup per message. The system can be dynamically adapted to the application behavior, learning from the previous communication history, and taking into account the specific communication characteristics (datatype, message size, etc), the platform specifications (network latency and bandwidth), the compression technique performance, and a threshold message size. We have integrated our system as a library in *MPICH*, generating a modified implementation called *Adaptive-CoMPI*. This implementation can be easily applied to any MPI implementation.

  Adaptive-CoMPI provides two decision strategies, Guided (GS) and Runtime Adaptive Compression (RAS) strategy, in order to decide as soon as possible whether to compress or not, and the best compression algorithm for each message. These strategies allow enabling and disabling compression on runtime. One of the major characteristics of RAS is that it can dynamically adapt itself to the application behaviour. This strategy learns from previous messages the compression algorithm that has been used and the size from which it obtains a benefit by compressing data. GS is thought to avoid RAS overhead, reducing execution time by providing application decision rules captured off-line based on the results obtained for each message. In addition, in Section 6.5 we have shown the improvement of activating and deactivating compression (Adaptive-CoMPI) over applying compression to all messages with the same compression algorithm (CoMPI).

  In this thesis, we have developed two models (network and compression behavior) in order to study the compression algorithms and network characteristics. First, using synthetic traces, we analyzed the performance (both in terms of compression ratio and execution time) of each compression technique. Different factors (type of data, buffer sizes, redundancy levels and data patterns) were considered, generating an output file with the best compression algorithms for each datatype. Second, we have measured the latency and bandwidth in our cluster by using MPI primitives in order to estimate the time to transmit a message. The output of these models are two heuristic files (network and compression) used by *Adaptive-CoMPI* to decide, for each message, the compression algorithm to be used.

  Evaluation results (Section 6.5) show that *Adaptive-CoMPI* can improve the speedups for many applications. They also demonstrate that, even in the worst cases, application performance is never reduced due to the overhead of our solution or the unfeasibility of finding an appropriate compressor. The re-

sults demonstrated that Runtime Adaptive Strategy performance is as good as Guided Strategy (off-line strategy). Furthemore the runtime performance gain is greater when more processes are employed, which increases scalability.

Performance results of *LA_ TwoPhase I/O* and *Adaptive-CoMPI* show considerable improvement for the communications and I/O subsytems. In addition, we have also integrated both techniques in the same implementation of *MPI*, generating a modified implementation called *Dynamic-CoMPI*, which can be easily included into any MPI implementation.

Dynamic-CoMPI has been validated by using several MPI benchmarks and real-life applications. As shown, those techniques are independent of the applications. Thus, they can be transparently applied at run-time without introducing modification in the application structure or source code.

The evaluation in Section 6.6 shows that for all of the considered scenarios, the speedup of *Dynamic-CoMPI* is the speedup of *LA-Two-Phase I/O* plus the speedup of *Adaptive-CoMPI*. That means that both techniques are complementary. We emphasize that the higher number of processes the greater the applications speedup achieved by *Dynamic-CoMPI* in most of cases. This behavior is due to the increasing number of communications. Therefore, the improvement of the communication performance produces a greater impact on the overall application performance. Evaluation study also demonstrates that even in the worst cases, application performance is never reduced due to the overhead of our solution. Additional benefits of our approach is the reduction of the total communication time and the network contention, thus enhancing, not only performance, but also scalability.

As a main conclusion, we state that the techniques proposed in this Ph.D. thesis produce a great impact on the overall performance and enhance scalability of MPI-based applications. As a final remark we conclude that the objectives proposed in this Ph.D. thesis have been sucesfully achieved.

7.2 Future Work

We are currently working to develope the implementation of *Dynamic-CoMPI* in Open MPI and mpich2. Also, it would be interesting to use more applications with longer execution time, as CMA or CFD applications.

As future work, we want to modify the aggregator pattern strategy explained in this work to fit even better to multicore architecture. This means assigning each aggregator acording to the local data that each node stores, instead of each core.

Furthermore, we want develop a new compression strategy based on message distribution patterns. We will include more compression algorithms in our library, especially for double precision floating point values.

7.3 Main Contributions

This thesis has resulted in the following publications:

- Journals:
  - Rosa Filgueira, David E. Singh, Alejandro Calderón, Felix García Carballeira and Jesús Carretero: Adaptive CoMPI: Enhancing MPI based applications performance and scalability by using adaptive compression. *International Journal of High Performance Computing and Applications, 2010.* [FCS+10a]
  - Rosa Filgueira, Jesús Carretero, David E. Singh, Alejandro Calderón and Alberto Nuñez: Dynamic-CoMPI: Dynamic optimization techniques for MPI parallel applications. *The Journal of Supercomputing, 2010.* [FCS+10b]
- Conferences:
  - Rosa Filgueira, David E. Singh, Alejandro Calderón and Jesús Carretero: CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression. *EUROPVM/MPI 2009* , pages: 207-218. [FSCC09]
  - Rosa Filgueira, David E. Singh, Juan Carlos Pichel and Jesús Carretero: Exploiting data compression in collective I/O techniques. *CLUSTER 2008*, pages: 479-485. [FSPC08]
  - Rosa Filgueira, David E. Singh, Juan Carlos Pichel, Florin Isaila and Jesús Carretero: Data Locality Aware Strategy for Two-Phase Collective I/O. *VECPAR 2008,* pages: 137-149. [FSP+08]
  - Rosa Filgueira, David E. Singh, Florin Isaila, Jesús Carretero and Antonio García Loureiro: Mejora de la localidad en operaciones de E/S colectivas no contiguas. Jornadas de Paralelismo. *CEDI 2007.* [FSCL07]
  - Rosa Filgueira, David E. Singh, Florin Isaila, Jesús Carretero and Antonio García Loureiro: Optimization and evaluation of parallel I/O in BIPS3D parallel irregular application. *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, pages: 1-8. [FSI+07]
- Grant:
  - Poster:
    - Rosa Filgueira, David E. Singh, and Jesús Carretero: Collective I/O Techniques for Chip Multiprocessor Clusters. IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2008.

Bibliography

[AB97] Arnold, Ross and Tim Bell: *A Corpus for the Evaluation of Lossless Compression Algorithms*. Data Compression Conference, 0:201, 1997.

[AMI09] Arif, Abu Shamim Mohammad, Asif Mahamud, and Rashedul Islam: *An Enhanced Static Data Compression Scheme Of Bengali Short Message*. CoRR, abs/0909.0247, 2009.

[BB99] Baker, Mark and Rajkumar Buyya: *Cluster Computing at a Glance*, 1999.

[BHS+95] Bailey, David, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow: *The NAS Parallel Benchmarks 2.0*. Technical Report 2, NASA, 1995.

[Bla86] Blackman., S.S.: *Multiple-Target Tracking with Radar Applications*. In *Dedham,MA: Artech House*, 1986.

[Bor97] Bordawekar, Rajesh: *Implementation of collective I/O in the intel paragon parallel file system: initial experiences*. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 20–27, New York, NY, USA, 1997. ACM.

[BR07] Burtscher, Martin and Paruj Ratanaworabhan: *High Throughput Compression of Double-Precision Floating-Point Data*. In *DCC07: Proceedings of the 2007 Data Compression Conference*, pages 293–302, Washington, DC, USA, 2007. IEEE Computer Society.

[BR09] Burtscher, Martin and Paruj Ratanaworabhan: *FPC: A High-Speed Compressor for Double-Precision Floating-Point Data*. IEEE Transactions on Computers, 58(1):18–31, 2009.

[BTR03] Balkanski, Daniel, Mario Trams, and Wolfgang Rehm: *Heterogeneous Computing With MPICH/Madeleine and PACX MPI: A Critical Comparison*, 2003.

[BYV08] Buyya, Rajkumar, Chee Shin Yeo, and Srikumar Venugopal: *Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities, in*. In *Department of Computer Science and*

Software Engineering (CSSE), The University of Melbourne, Australia. He, pages 10–1016, 2008.

[CFS02] CFS Inc.: *Lustre: A scalable, high-performance file system*, 2002. Cluster File Systems Inc. white paper, version 1.0,.

[CNP+98] Carretero, J., J. No, S. S. Park, A. Choudhary, and P. Chen: *COMPASSION: a parallel I/O runtime system including chunking and compression for irregular applications*. In *Proceedings of the International Conference on High-Performance Computing and Networking*, pages 668–677, April 1998.

[Com07] Company, Hewlett Packard: *HP-MPI Users Guide, 11th Edition*, 2007.

[CP88] Carpaneto, S.M. Giorgio and P.Oth.: *Algorithms and codes for the assignment problem*. Annals of Operations Research, 13(1):191–223, 1988.

[Dav94] David Kotz: *Disk-directed I/O for mimd multiprocessors*. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, 1994.

[Dav04] Davies, Antony: *Computational intermediation and the evolution of computation as a commodity*. Applied Economics, 36(11):1131–1142, 2004.

[DBA+09] Dongarra, Jack, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo Valero: *The International Exascale Software Project: a Call To Cooperative Action By the Global High-Performance Community*. Int. J. High Perform. Comput. Appl., 23(4):309–322, 2009.

[DL08] Dickens, Phillip and Jeremy Logan: *Towards a High Performance Implementation of MPI-IO on the Lustre File System*. In *OTM 08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems*, pages 870–885, Berlin, Heidelberg, 2008. Springer-Verlag.

[DLY+98] Davis, G., L. Lau, R. Young, F. Duncalfe, and L. Brebber: *Parallel run-length encoding (RLE) compression—reducing I/O in dynamic environmental simulations*. The International Journal of High Performance Computing Applications, 12(4):396–410, Winter 1998. In a Special Issue on I/O in Parallel Applications.

[Ead06] Eadlin, Douglas: *Preparing for the Revolution Maximizing Dual Core Technology*, 2006.

[Eis09] Eisner, Jason: *Cloud Computing: Business Benefits With Security, Governance and Assurance Perspectives*, 2009. Available online (10 pages).

[FCS+10a] Filgueira, Rosa, Jesus Carretero, David E. Singh, Alejandro Calderon, and Felix Garcia: *Adpative-CoMPI: Enhancing MPI based applications performance and scalability by using adaptive compression*, April 2010. International Journal of High Performance Computing and Applications.

[FCS+10b] Filgueira, Rosa, Jesus Carretero, David E. Singh, Alejandro Calderon, and Alberto Nunez: *Dynamic-CoMPI: Dynamic optimization techniques for MPI parallel applications*. The Journal of Supercomputing, 0(0), 2010.

[FK96] Foster, Ian and Carl Kesselman: *Globus: A Metacomputing Infrastructure Toolkit*. International Journal of Supercomputer Applications, 11:115–128, 1996.

[FK99] Foster, Ian and Carl Kesselman: *The globus toolkit*. pages 259–278, 1999.

[Flo08] Florin Daniel Isaila: *ClusterFile: A Parallel File System for Clusters*, 2008. PhD thesis, Universitat Karlsruhe (Technische Hochschule).

[FSCC09] Filgueira, Rosa, David E. Singh, Alejandro Calderon, and Jesus Carretero: *CoMPI:Enhancing MPI based applications performance and scalability using compression*. European PVM/MPI, 2009.

[FSCL07] Filgueira, Rosa, David E. Singh, Jesus Carretero, and Antonio Garcia Loureiro: *Mejora de la localidad en operaciones de E/S colectivas no contiguas*. In *CEDI 2007*, page 0, 2007.

[FSI+07] Filgueira, Rosa, David E. Singh, Florin Isaila, Jesus Carretero, and Antonio Garcia Loureiro: *Optimization and evaluation of parallel I/O in BIPS3D parallel irregular application*. In *IPDPS*, pages 1–8, 2007.

[FSP+08] Filgueira, Rosa, David E. Singh, Juan Carlos Pichel, Florin Isaila, and Jesus Carretero: *Data Locality Aware Strategy for Two-Phase Collective I/O*. In *VECPAR*, pages 137–149, 2008.

[FSPC08] Filgueira, Rosa, David E. Singh, Juan Carlos Pichel, and Jesus Carretero: *Exploiting data compression in collective I/O techniques*. In *CLUSTER*, pages 479–485, 2008.

[GCC99] Garcia, Felix, A. Calderon, and Jesus Carretero: *Mimpi: A multithread-safe implementation of mpi*. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 207–214. 6th PVM/MPI European Users Group Meeting, 1999.

[GDV94] G. Burns, R. Daoud, and J. Vaigl: *LAM: An open cluster environment for MPI*. Proceedings of Supercomputing Symposium '94, 1994.

[GFB+04] Gabriel, Edgar, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall: *Open MPI: Goals, concept, and design of a next generation MPI implementation.* In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GL97] Gropp, W. and E. Lusk: *Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing.* The International Journal of Supercomputer Applications and High Performance Computing, 11(2):103–114, 1997.

[GL99] Gropp, William and Ewing L. Lusk: *Reproducible Measurements of MPI Performance Characteristics.* In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, 1999. Springer-Verlag.

[GLDS96] Gropp, W., E. Lusk, N. Doss, and A. Skjellum: *A high-performance, portable implementation of the MPI message passing interface standard.* Parallel Computing, 22(6):789–828, September 1996.

[Gol] Golomb, RS.W. *IEEE Trans. Information Theory.*

[GVdB01] Goeman, Bart, Hans Vandierendonck, and Koen de Bosschere: *Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency.* In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[HC06] Hastings, Andrew B. and Alok N. Choudhary: *Exploiting Shared Memory to Improve Parallel I/O Performance.* In *PVM/MPI*, 2006.

[HM93] Herlihy, Maurice and J. Eliot B. Moss: *Transactional Memory: Architectural Support for Lock-Free Data Structures.* In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[Hoc94] Hockney, Roger W.: *The communication challenge for MPP: Intel Paragon and Meiko CS-2.* Parallel Computing, 20(3):389–398, 1994.

[HSJ+06] Huang, W., G. Santhanaraman, H. W. Jin, Q. Gao, and D. K. x. D. K. Panda: *Design of High Performance MVAPICH2: MPI2 over InfiniBand.* In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 43–48, Washington, DC, USA, 2006. IEEE Computer Society.

[I-W95] *Virtual Environments and Distributed Computing at SC'95 : GII Testbed and HPC Challenge Applications on the I-WAY*, 1995.

[IEE90] IEEE: *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE, 1990.

[KBI01] Krauter, Klaus, Klaus Krauter I Rajkumar Buyya, and Muthucumaru Maheswaran It: *A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing*, 2001.

[KBS04a] Ke, Jian, Martin Burtscher, and Evan Speight: *Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications*. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 59, Washington, DC, USA, 2004. IEEE Computer Society.

[KBS04b] Ke, Jian, Martin Burtscher, and Evan Speight: *Runtime Compression of MPI Messanes to Improve the Performance and Scalability of Parallel Applications*. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 59, 2004.

[KCJ+95] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett: *Server-directed collective I/O in panda*. Proceedings of Supercomputing '95, 1995.

[Kes98] Kesselman, Carl and Foster, Ian: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[KHQ+94] Kowalik, Janusz, Philip J. Hatcher, Michael J. Quinn, Edited Piyush Mehrotra, Joel Saltz, Robert Voigt, Edited Horst D. Simon, Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Mary E. Zosel, Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, and Vaidy Sunderam: *PVM: Parallel Virtual Machine*, 1994.

[KK98] Karypis, George and Vipin Kumar: *MeTiS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Technical report, 1998.

[KKZ07] Kennedy, Ken, Charles Koelbel, and Hans Zima: *The rise and fall of high performance fortran: an historical object lesson*. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[KNTG08] Kumar, V. Santhosh, R. Nanjundiah, M. J. Thazhuthaveetil, and R. Govindarajan: *Impact of message compression on the scalability of an atmospheric modeling application on clusters*. Parallel Comput., 34(1):1–16, 2008.

[Knu85] Knuth, Donald E.: *Dynamic Huffman coding*. J. Algorithms, 6(2):163–180, 1985.

[lG10] Gailly, Jean loup: *Gzip compressor*, 2010. "http://www.gzip.org/".

[LGT03] Loureiro, A.J.García, J.M.López González, and T.F.Pena: *A parallel 3D semiconductor device simulator for gradual heterojunction bipolar transistors*. Int. Journal of Numerical Modelling: electronic networks, devices and fields, 16:53–66, 2003.

[LI06] Lindstrom, Peter and Martin Isenburg: *Fast and Efficient Compression of Floating-Point Data*. IEEE Transactions on Visualization and Computer Graphics, 12(5):1245–1250, 2006.

[LIR99] Ligon, Walter B., III, and Robert B. Ross: *An Overview of the Parallel Virtual File System*. In *Proceedings of the 1999 Extreme Linux Workshop*, 1999.

[LMP03] Liu, Jiuxing, Amith R Mamidala, and Dhabaleswar K Panda: *Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support*. In *In Proceedings of IPDPS*, 2003.

[Maj96] Majumdar, Amitava: *On Evaluating and Optimizing Weights for Weighted Random Pattern Testing*. IEEE Trans. Comput., 45(8):904–916, 1996.

[Mar02] Markus Franz Xaver Johannes Oberhumer: *LZO*, 2002. http://www.oberhumer.com/opensource/lzo/lzodoc.php.

[Mes94] Message Passing Interface Forum: *MPI: A message-passing interface standard*. International Journal of Supercomputer Applications, 8:165–414, 1994.

[MIT10] MIT: *Cilk programming language*, 2010. "http://supertech.csail.mit.edu/cilk/".

[MLM98] Mucci, Phillip J., Kevin London, and Philip J. Mucci: *The MPBench Report*. Technical report, 1998.

[MM05] Martinasso, Maxime and Jean François Méhaut: *Prediction of Communication Latency over Complex Network Behaviors on SMP Clusters*. In *EPEW/WS-FM*, pages 172–186. IEEE Computer Society, 2005.

[MMD+04] Mourino, J., M. Martin, R. Doallo, D. Singh, F. Rivera, and J. Bruguera: *The STEM-II air quality model on a distributed memory system*, 2004.

[Moh09] Mohr, Bernd: *Summary of the IESP White Papers*. IJHPCA, 23(4):323–327, 2009.

[MSM+01] Mourino, Jose Carlos, David E. Singh, Maria J. Martin, J. M. Eiroa, Francisco F. Rivera, Ramon Doallo, and Javier D. Bruguera: *Parallelization of the STEM II Air Quality Model*. In *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 543–546, London, UK, 2001. Springer-Verlag.

[MVCA97] Martin, Richard P., Amin M. Vahdat, David E. Culler, and Thomas E. Anderson: *Effects of communication latency, overhead, and bandwidth in a cluster architecture*. SIGARCH Comput. Archit. News, 25(2):85–97, 1997.

[NAS] NASA: *NAS Parallel Benchmarks*. http://www.nas.nasa.gov/NAS/NPB.

[NKP+96a] Nieuwejaar, Nils, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best: *File-access characteristics of parallel scientific workloads*. IEEE Transactions on Parallel and Distributed Systems, 7(10):1075–1089, 1996.

[NKP+96b] Nieuwejaar, Nils, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best: *File-Access Characteristics of Parallel Scientific Workloads*, 1996.

[NKP+96c] Nieuwejaar, Nils, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best: *File-Access Characteristics of Parallel Scientific Workloads*. IEEE Transactions on Parallel and Distributed Systems, 7(10):1075–1089, 1996.

[Obe05] Oberhumer, M. F. X. J.: *LZO real-time data compression library*, 2005.

[OCH+08] Ou, Li, Xin Chen, He, X; Engelmann, Christian, Scott, and Steven L: *Achieving computational I/O efficiency in a high performance cluster using multicore*. 2008.

[Ope05] Openmp: *OpenMP Architecture Review Board, OpenMP Application Program Interface, version 2.5*, 2005.

[P. 97] P. Bhargava: *Mpi-lite user manual, release 1.1*. Technical report, Parallel Computing Lab, University of California, 1997.

[Phi00] Phil Merkey: *Beowulf history*, 2000. Beowulf.org (http://www.beowulf.org/beowulf/history.html).

[PSR06] Pichel, Juan C., David E. Singh, and Francisco F. Rivera: *Image segmentation based on merging of sub-optimal segmentations*. Pattern Recogn. Lett., 27(10):1105–1116, 2006.

[Qua02] Quammen, Cory: *Introduction to programming shared-memory and distributed-memory parallel computers*. Crossroads, 8(3):16–22, 2002.

[RA87] R.Jonker and A.Volgenant.: *A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems*. Computing, 38(4):325–340, 1987.

[RA08] Rashti, Mohammad J. and Ahmad Afsahi: *Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects*. In *HPCS '08: Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, pages 95–101, Washington, DC, USA, 2008. IEEE Computer Society.

[Raj99] Rajkumar Buyya: *High Performance Cluster Computing (Volumen 1)*. Prentice Hall, 1999.

[RBMA94] R. Alasdair, A. Bruce, J.G. Mills, and Smith A.G.: *CHIMP-MPI user guide*. Technical report, Edinburgh Parallel Computing Centre, Edinburgh Parallel Computing Centre, 1994.

[RC95] Rajeev Thakur and Alok Choudhary: *An extended two-phase method for accessing sections of out-of-core arrays*. Technical Report CACR-103, Center for Advanced Computing Research, 1995.

[Rei07] Reinders, James: *Intel threading building blocks*. OReilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[RGL99] Rajeev Thakur, William Gropp, and Ewing Lusk: *Data sieving and collective I/O in ROMIO*. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. Argonne National Laboratory, 1999.

[RK05] R. Keller, M. Liebing: *Using PACX-MPI in metacomputing applications*. In *18th Symposium Simulationstechnique*, Erlangen, September 12-15, 2005.

[RKB06] Ratanaworabhan, Paruj, Jian Ke, and Martin Burtscher: *Fast Lossless Compression of Scientific Floating-Point Data*. In *DCC '06: Proceedings of the Data Compression Conference*, pages 133–142, Washington, DC, USA, 2006. IEEE Computer Society.

[ROM] *Romio: A high-performance, portable MPI-IO implementation*. ROMIO (http://www.mcs.anl.gov/romio).

[RST02] Reussner, Ralf, Peter Sanders, and Jesper Larsson Träff: *SKaMPI: a comprehensive benchmark for public benchmarking of MPI*. Sci. Program., 10(1):55–65, 2002.

[SC00] Salvatore Coco, Valentina D'Arrigo, Domenico Giunta: *A Rice-based Lossless Data Compression System For Space*. In *In proceedings of the 2000 IEEE Nordic Signal Processing Symposium*, pages 133–142, 2000.

[SGC05] Singh, David E., Felix Garcia, and Jesus Carretero: *Parallel I/O Optimization for an Air Pollution Model*. In *PARCO*, pages 523–530, 2005.

[SH02] Schmuck, Frank and Roger Haskin: *GPFS: A shared-disk file system for large computing clusters*. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[SHL+06] Shi, Zhongzhi, He Huang, Jiewen Luo, Fen Lin, and Haijun Zhang: *Agent-based grid computing*. Applied Mathematical Modelling, 30(7):629 – 640, 2006. Parallel and Vector Processing in Science and Engineering.

[SR98] Simitci, Huseyin and Daniel A. Reed: *A comparison of logical and physical parallel I/O patterns*. The International Journal of Supercomputer Applications and High Performance Computing, 12(3):364–380, Fall 1998.

[SS97] Sazeides, Yiannakis and James E. Smith: *The predictability of data values*. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[Sym00] Symes, Peter: *Video Compression Demystified*. McGraw-Hill Professional, 2000.

[vdS09] Steen, Aad J. van der: *Overview of recent supercomputers*. Technical report, Knoxville, TN, USA, 2009.

[Vis01] *Visual KAP for OpenMP*, 2001. http://www.kai.com/vkomp/_index.html.

[VSRC95] Vaughan, P. L., A. Skjellum, D. S. Reese, and Fei Chen Cheng: *Migrating from pvm to mpi.i. the unify system*. In *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, page 488, Washington, DC, USA, 1995. IEEE Computer Society.

[Wel05] Wells, George: *Coordination Languages: Back to the Future with Linda*. In *Proceedings of WCAT05*, pages 87–98, 2005.

[WSH06] Werstein, Paul, Hailing Situ, and Zhiyi Huang: *Load Balancing in a Cluster Computer*. Parallel and Distributed Computing Applications and Technologies, International Conference on, 0:569–577, 2006.

[Wun90] Wunderlich, H. J.: *Multiple distributions for based random test patterns*. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, 9(6):584–593, 1990.

[YVCJ07] Yu, Weikuan, Jeffrey Vetter, R. Shane Canon, and Song Jiang: *Exploiting Lustre File Joining for Effective Collective I/O*. Cluster Computing and the Grid, IEEE International Symposium on, 0:267–274, 2007.

[Zig89] Zigon, Robert: *Run length encoding.* Dr. Dobb's Journal of Software Tools, 14(2), February 1989.